9 Celebrity Compilers

9.1 Introduction

Here we survey some of the popular Java (or Java-like) compilers. For each of these, we discuss issues or features that are peculiar to the compiler in question. The compilers we discuss are the following.

- Oracle's Java HotSpot™ compiler.
- IBM's Eclipse compiler for Java.
- The GNU Java compiler.
- Microsoft's C# compiler.

This chapter will only give a taste of these compilers. The reader may wish to consult some of the recommended readings in the Further Reading section at the end of this chapter for a deeper understanding of them.

9.2 The Java HotSpot™ Compiler

The original Java Virtual Machine (JVM) was developed by Sun Microsystems. The Oracle Corporation, who acquired Sun in 2010, now maintains it. What makes Oracle's JVM compiler (Oracle, 2011) special, aside from the fact that it is *the* original Java compiler, is its implementation of its just-in-time (JIT) compiler.

A typical JIT compiler will translate a method's (JVM) byte code into native code the first time the method is invoked and then cache the native code. In this manner, only native code is executed. When a method is invoked repeatedly, its native code may be found in the cache. In programs with large loops or recursive methods, JIT compilation drastically reduces the execution time (Kazi et al, 2000).

An obvious drawback of JIT compilation is the initial run-time delay in the execution of a program, which is caused by both the compilation of methods the first time they are encountered and the application of platform dependent optimizations. JIT compilers are able to perform optimizations that static compilers[1] cannot, since they have access to run-time information, such as input parameters, control flow and target machine specifics (e.g. the compiler knows what processor the program is running on and can tune the generated code accordingly). Other than that, JIT compilers do not spend as much time on optimizing as the static compilers do since the time spent on optimizations adds to the execution time. Also, because classes can be dynamically invoked or each method is compiled on demand, it is difficult for JIT compilers to perform global optimizations.

[1] Static compilation is also known as Ahead-of-Time Compilation, which is discussed in Section 9.4.2.

When compared to interpretation, JIT compilation has another downside; in JIT compilation, methods are entirely compiled before they are executed whereas in interpretation only the instructions actually to be executed are translated. If only a small part of a large method is executed, JIT compilation may be time-consuming (Kazi et al, 2000). The memory usage is increased as well since native code requires more space than the more compact JVM byte code.

As we saw in Chapter 7, the Oracle JVM uses a JIT compilation regime called HotSpot. Knuth's finding, which states that most programs spend the majority of time executing a small fraction of the code (Knuth, 1971), lays the foundation of a technique called *adaptive optimization*.

The Java HotSpot VM has both an interpreter and a byte code to native machine code compiler. Instead of compiling every method it encounters, the VM first runs a profiling interpreter to quickly gather runtime information, detect the critical "hot spots" in the program which are executed most often and collect information about the program behavior so it may use it to optimize the generated native code in later stages of program execution. The identified hot spots are then compiled into optimized native code, while infrequently executed parts of the program continue to be interpreted. As a result, more time can be spent on optimizing those performance-critical hot spots, and the optimizations are smarter than static compiler optimizations because of all the information gathered during interpretation. Hot spot monitoring continues during run time to adapt the performance according to program behavior.

There are two versions of the Java HotSpot VM: the Client VM and the Server VM. These two VMs are identical in their runtime, interpreter, memory model and garbage collector components; they differ only in their byte code to native machine code compilers.

The client compiler is simpler and focuses on extracting as much information as possible from the byte code, like locality information and an initial control flow graph, to reduce the compilation time. It aims to reduce the initial start-up time and memory footprint on users' computers.

The server compiler on the other hand, is an advanced dynamic optimizing compiler focusing on the execution speed. It uses an advanced SSA-based IR[2] for optimizations. The optimizer performs classic optimizations like dead code elimination, loop invariant hoisting, common sub-expression elimination and constant propagation, as well as Java-specific optimizations as null-check and range check elimination[3]. The register allocator

[2] SSA stands for "Static Single Assignment" and IR stands for "Intermediate Representation". See Chapter 7 for detailed explanation.
[3] See Chapter 7 for various optimizations techniques.

is a global graph coloring allocator[4] (Oracle Arch., 2010). However, the most important optimizations performed by the Java HotSpot server compiler are *method inlining* and *dynamic de-optimization*.

Method inlining is a common optimization technique, which replaces a method's call in the code with the corresponding method's body. This removes the cost of call and return overhead, and facilitates optimizations on the replacement code. Method inlining leads to larger basic blocks that are more amenable to optimization.

Static compilers cannot take full advantage of this optimization technique since they cannot know if a method is overridden in a subclass. A static compiler can conservatively inline static, final and private methods since such methods cannot be overridden, but there is no such guarantee for public and protected methods. Moreover, classes can be loaded dynamically during runtime and a static compiler cannot deal with such run-time changes in the program structure (Wilson and Kesselman, 2000).

While method inlining can reduce both compilation time and execution time, it will increase the total size of the produced native machine code. That is why it is important to apply this optimization selectively rather than blindly inlining every method call. The HotSpot compiler applies method inlining in the detected program hot spots. HotSpot has the freedom of inlining any method since it can always undo an inlining if the method's inheritance structure changes during runtime.

Undoing the inlining optimizations is called dynamic de-optimization. Suppose the compiler comes across to a virtual function call as

```
MyProgram.foo();
```

There are two possibilities here; the compiler can run the **`foo()`** function implemented in **`MyProgram`**, or a child class's implementation of **`foo()`**. If **`MyProgram`** is defined as a final class, or **`foo()`** in **`MyProgram`** is defined as a final method, the compiler knows that **`MyProgram`**'s **`foo()`** should be executed. What if that is not the case? Well, then the compiler has to make a guess, pick whichever seems right to it at the moment. If it is lucky, there is no need to go back; but that decision may turn out to be wrong or invalidated by a class loaded dynamically that extends **`MyProgram`** during runtime (Goetz, 2004). Every time a class is dynamically loaded, the HotSpot VM checks whether the interclass dependencies of in-lined methods have been altered. If there are any altered dependencies, the HotSpot VM can dynamically de-optimize the affected in-lined code, switch to interpretation mode and maybe re-optimize later during runtime as a result of new class dependencies. Dynamic de-optimization allows HotSpot VM to perform inlining aggressively with the confidence that possible wrong decisions can always be backed out.

[4] See Chapter 8 for register allocation techniques, including graph-coloring allocation.

If we look at the evolution of Oracle's JVM since the beginning, we see that it has matured in stages. Early VMs always interpreted byte code, which could result in 10 to 20 times slower performance compared to C. Then, a JIT compiler was introduced for the first time with JDK 1.2 as an add-on, and with JDK 1.3 HotSpot became the default compiler. That is when major performance improvements were achived. When compared to the initial release, JDK 1.3 showed a 40% improvement in start-up time and a 25% smaller RAM footprint (Shudo, 2005). Later releases improved HotSpot even more. Thus, the performance of a program written in Java code today approaches that of an equivalent C program.

As an example of improvements made to HotSpot, consider the history of the JVM's handling of the transfer of control between byte code and native code. In HotSpot's initial version, two counters were associated with each method: a method-entry counter and a backward-branch counter. The method entry counter was incremented every time the method was called while the backward-branch counter was incremented every time a backward branch was taken (e.g. imagine a for-loop; the closing brace of the for-loop's body would correspond to a backward branch); and these counters were combined into a single counter value with some additional heuristics for the method. Once the combined counters hit a threshold (10,000 in HotSpot version 1.0) during interpretation, the method would be considered "hot" and it then would be compiled (Paleczny et al, 2001). However, HotSpot compiler could only execute the compiled code for this method when the method is called next time.

In other words, if the HotSpot compiler detected a big, computationally intensive method as a hot-spot and compiled it into native code say, because of a loop at the beginning of the method, it would not be able to use the compiled version of the method until the next time it was invoked. It was possible that heavyweight methods were compiled during runtime but the compiled native code would never be used (Goetz, 2004). Programmers used to "warm-up" their programs for HotSpot by adding redundant and very long running loops in order to provoke the HotSpot to produce native code at specific points in their program and obtain better performance at the end.

On-stack replacement (OSR) is used to overcome this problem. It is the opposite of dynamic de-optimization, where the JVM can switch from interpretation mode to compilation mode or swap a better version of compiled code in the middle of a loop, without waiting for the enclosing method to be exited and re-entered (Oracle OSR, 2010). When the interpreter sees a method looping, it will invoke HotSpot to compile this method. While the interpreter is still running the method, HotSpot will compile the method aside, with an entry point at the target of the backward branch. Then the runtime will replace the interpreted stack activation frame with the compiled frame, so that execution continues with the new frame from that point (Paleczny et al, 2001). HotSpot can apply aggressive specializations based on the current conditions at any time and re-

compile the code that is running if those conditions change during runtime by means of OSR.

Clever techniques like aggressive inlining, dynamic de-optimization, OSR and many others (Paleczny et al, 2001 and Oracle, 2011) allow HotSpot to produce better code when compared to traditional JIT compilation and make it a very interesting compiler to study.

9.3 The Eclipse Compiler for Java (ECJ)

Eclipse is an open source development platform comprised of extensible frameworks, tools and runtimes. The Eclipse platform (Eclipse, 2011) is structured as a set of subsystems, which are implemented as plug-ins. The Eclipse Java Development Tools (JDT) provide the plug-ins for a Java integrated development environment with its own Java compiler called the *Eclipse Compiler for Java* (ECJ), which is often compared with Oracle's javac.

ECJ is an incremental compiler, meaning that after the initial compilation of a project, it compiles only the modified (or newly added) file and its dependent files next time, instead of re-compiling the whole project again.

Compilation in Eclipse can be invoked in three different ways (Eclipse, 2004):

- **Full Compilation** requires that all source files in the project should be compiled. This is either performed as the initial compilation, or as a result of a *clean* operation that is performed on the project, which deletes all the `.class` files and problem markers, thus forcing a recompilation of an entire project.

- **Incremental Compilation** compiles only the changed files (by visiting the complete resource delta tree) and the files that are affected by the change, e.g. classes that implement an interface, the classes calling methods of the changed classes, etc.

- **Auto Compilation** is essentially the same as incremental compilation. Here, incremental compilation is triggered automatically because a change in source code has occurred.

Eclipse's incremental compiler for Java uses a "last build state" to do an optimized build based on the changes in the project since the last compilation. The changes since the last compilation are captured as a *resource delta tree*, which is a hierarchical description of what has changed between two discrete points in the lifetime of the Eclipse workspace. The next time ECJ is called, it uses this resource delta tree to determine the source files that need to be recompiled because they were changed, removed, or added.

In addition to the resource delta tree, the compiler keeps a *dependency graph* in memory as part of its *last built state*. The dependency graph includes all the references from each type to other types. It is created from scratch in the initial compilation, and updated incrementally with new information on each subsequent compilation. By using this graph, the compiler can decide if any structural changes[5] occur as a consequence of changing, adding or deleting a file. Computation of these structural changes constitutes the set of source files that might compile differently as a consequence. The compiler deletes obsolete class files and associated Java problem markers that were added to the source files due to compile errors previously, and compiles only the computed subset of source files. The dependency graph is saved between sessions with workspace saves. The dependency graph does not have to be re-generated and the compiler avoids full compilation every time the project is opened. Of course, the last built state should be updated with the new reference information for the compiled type, and new problem markers should be generated for each compiled type if it has any compilation problems, as the final steps of compilation (Eclipse, 2006). Figure 9.1 shows these steps:

Incremental compilation is very efficient, especially in big projects with hundreds of source files, since most of the source files will remain unchanged between two consecutive compilations. Hence, frequent compilations on hundreds or thousands of source files can be performed without delay.

Moreover, most times when a Java file is changed, it does not result in any structural changes meaning that there is only a single file to be compiled. Even when structural changes occur and all referencing types need to be recompiled, those secondary types will almost never have structural changes themselves, so the compilation will be completed with at most a couple of iterations. Of course, one cannot claim that there will never be significant structural changes that may cause many files to be recompiled. ECJ considers this tradeoff worth the risk, assuming if the compiler runs very fast for the most common cases; rare occasions of longer delays are acceptable to users.

[5] Structural changes are the changes that can affect the compilation of a referencing type (e.g. added or removed methods, fields or types, or changed method signatures).

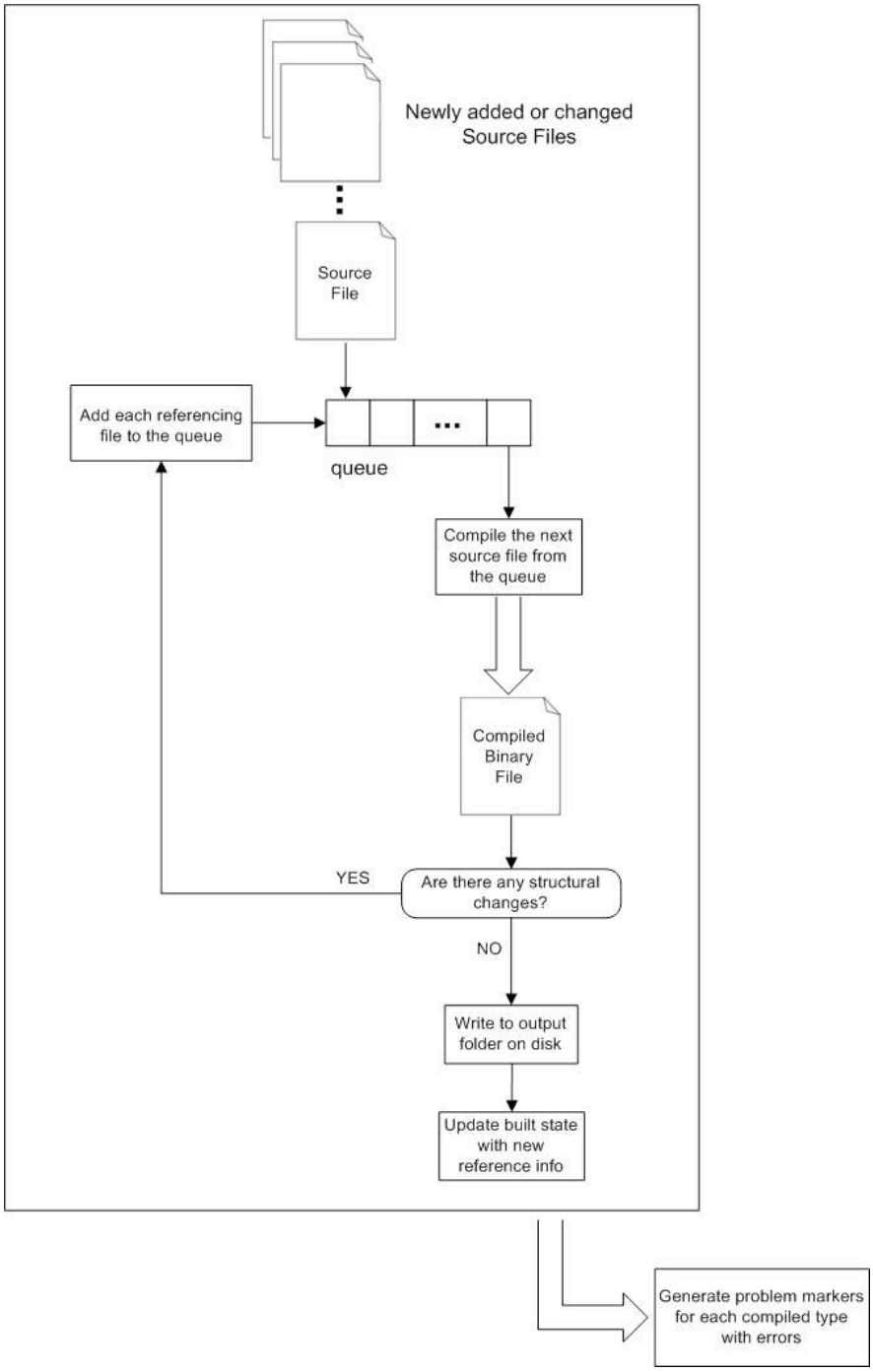

Figure 9.1 Steps to ECJ Incremental Compilation

Incremental compilers are usually accused of not optimizing enough because of the locality and relatively small amount of the code that has been changed since the last build. However, this is not a real problem, since all the heavy weight optimizations are performed during runtime. Java compilers are not expected to optimize heavily when

compiling from source code to bytecode. ECJ performs light optimizations like discarding unused local variables from the generated bytecode and inlining to load the bytecode faster into the VM (since the verification process is then much simpler even though the code size to load increases), but the real runtime performance difference comes from the VM that is used (e.g. IBM's JVM, Oracle's JVM, etc.). And if one decides to use, say, Oracle's JVM, choosing the latest version would be a good practice since HotSpot is being improved in every new version.

Perhaps one of the most interesting features of ECJ is its ability to run and debug code that contains errors (Eclipse, 2011). Consider the following naïve test.

```
public class Foo
{
	public void foo()
	{
		System.println("I feel like I forgot something...");
	}

	public static void main(String[] args)
	{
		System.out.println("It really works!");
	}
}
```

ECJ will flag the erroneous use of `println()` within `foo()`, marking the line in the editor and underlining the `println`; however, it will still generate the bytecode. And when the generated bytecode is executed, it will run properly since `foo()` is never called in the program. If the control reaches to the error, the runtime will then throw an exception. This feature is useful when testing individual complete pieces of a project that contains incomplete pieces.

A lesser-known feature of ECJ is its support for "Java Scrapbook Pages". A scrapbook page allows programmers to test small pieces of Java expressions without creating a surrounding Java class or method for the specific code snippet.

A scrapbook page can be created by either simply creating a file with .jpage extension, or using the *New Java Scrapbook Page* wizard in Eclipse (Eclipse, 2010). One can write a single line of code as

```
new java.util.Date()
```

inside the new Scrapbook page, and then display the result of evaluating this expression by highlighting it and clicking the *Display* button. The output is displayed within the scrapbook editor alongside the expression. Notice, a semi-colon at the end of line is omitted, which would normally cause a compile-time error in a java file; yet ECJ

understands what is intended here and evaluates the expression. The scrapbook editor also supports code assist, showing the possible completions when writing a qualified name.

Another convenience is the ability to *Inspect* a highlighted expression, which will open a pop-up window inside the editor and show all the debug information on the highlighted object without having to switch to the Debug perspective.

There is also the ability to *Execute* scrapbook expressions on the JVM. Suppose a scrapbook page has the following lines.

```
java.util.Date now = new java.util.Date();
System.out.println("Current date and time: " + now);
```

Executing these two lines in scrapbook produce the requisite output:

```
Current date and time: Mon Sep 26 23:33:03 EDT 2011
```

The scrapbook provides that immediate feedback that Lisp users find so useful.

ECJ is also used in Apache Tomcat for compiling JSP pages (Apache Tomcat, 2006) and the Java IDE called IntelliJ IDEA (JetBrains, 2011). In addition, GCJ, discussed in the next section, uses ECJ as its front-end compiler (as of GCJ 4.3).

9.4 The GNU Java Compiler (GCJ)

9.4.1 Overview

GNU Java Compiler (GCJ) (GNU, 2011) is one of the compilers provided in the GNU Compiler Collection (GCC). Although GCC started as a C compiler (aka the GNU C Compiler), compilers for other languages including Java were soon added to the collection. Today, GCC is the standard compiler for many Unix-like operating systems.

GCJ may directly produce either native machine code or JVM byte-code class files. GCJ can deal with both Java source files or zip/jar archives, and can package Java applications into both .jar files and Windows executables (.exe files) as well as producing class files (GCJ, 2011).

GCJ uses the Eclipse Java Compiler (ECJ) as its front-end. Programs that are compiled into byte code with ECJ are linked with the GCJ runtime called libgcj. Libgcj provides typical runtime system components such as a class loader, core class libraries, a garbage

collector[6] and a bytecode interpreter. Libgcj can also interpret source code to machine code directly. Figure 9.2 shows possible routes a Java source code can take when fed to GCJ.

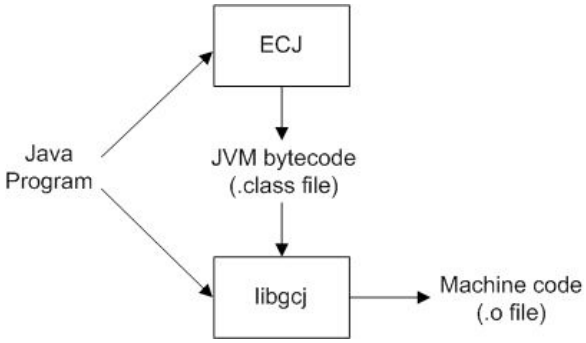

Figure 9.2 Possible paths a Java program take in GCJ

Compiled Java programs are next fed into the GCJ back-end to be optimized and executed. The GCJ back-end and the philosophy behind it are discussed in the next section.

9.4.2 GCJ in Detail

Other than to provide a Java implementation under the GPL license, a principal motivation for GCJ was to compete with JIT performance. As we saw in section 9.2, JIT compilation suffers from issues with start-up overhead and the overhead in detecting hot spots.

GCJ took a "radically traditional" approach to overcome these problems and followed the classical ahead-of-time (AOT) compilation, where byte-code programs are compiled into a system-dependent binary *before* executing the programs.

GCJ developers treated Java like another form of C++. Java programs are translated to the same abstract syntax tree used for other GCC source languages. Thus, all existing optimizations like common sub-expression elimination, strength reduction, loop optimization and register allocation that were available for GNU tools could be used in GCJ (Bothner, 2003).

As a result, programs compiled directly to native code using the GCC back-ends run at full native speed with low start-up overhead and modest memory usage (Wielaard, 2005). This makes GCJ a suitable compiler for embedded systems.

---

[6] Hans Boehm's conservative garbage collector, which is known as *Boehm GC*, is used. Detailed information can be found at http://www.hpl.hp.com/personal/Hans_Boehm/gc/

On the other hand, after start-up, Java programs compiled by GCJ do not necessarily always run faster than a modern JIT compiler like HotSpot. There are various reasons for this.

- First of all, some optimizations possible in JIT compilation, such as runtime profile-guided optimizations and virtual function inlining, are just not possible in GCJ.
- Programs that allocate many small, short-lived objects can cause the heap to fill quickly and garbage collected often, which in turn will slow down the execution.
- GCJ has sub-optimal runtime checking code, and the compiler is not so smart about automatically removing array checks. A (dangerous) hack for this can be compiling with GCJ's `--no-bounds-check` option.
- On many platforms, dynamic (PIC[7]) function calls are more expensive than static ones. In particular, the interaction with Boehm-garbage collector seems to incur extra overhead when shared libraries are used. In such cases, static linking[8] can be used to overcome this problem.
- GCJ does not behave well with threads. Multi-threaded processes are less efficient in GCJ.

A useful feature of GCJ is the flexibility it offers in choosing what optimizations to perform. GCJ categorizes optimizations by level and permits users to choose the level of optimizations they want to apply on the programs. There are five optimization levels, each specified by the `-Ox` (x representing the optimization level) option (GNU, 2010):

- The default optimization level is zero, meaning that no optimizations are applied. At this level, the compiler's goal is to reduce the cost of compilation.
- Level-1 aims to produce modestly optimized code in a short time. Optimizations of this level designed to reduce the size of the compiled code and increase the performance at the same time without taking too much of the compilation time.
- Level-2 optimizes the code more than level-1. This level supports the set of optimizations that do not involve a space-speed tradeoff.
- Level-s is for optimizing for size. (Some call it level 2.5.) This level basically enables all level-2 optimizations that do not increase the code size, and also applies additional optimizations that reduce the code size.

---

[7] Position-Independent Code (PIC) is machine instruction code that can be copied to an arbitrary location in memory and work without requiring any special processing based on its memory location. PIC is commonly used for shared libraries.

[8] Static linking refers to resolving the calls to libraries in a caller and copying them into the target application program at compile time, rather than loading them in at runtime.

- Level-3 is the highest level of optimization. The optimizations at this level focus on speed rather than size. However, one must be careful when using this level of optimizations because if the size of the optimized image exceeds the size of the available instruction cache, performance can be severely hindered too.

Other than using these optimization levels, one can enable particular optimizations explicitly by specifying their names.

Overall, GCJ is an eminent open-source option, for embedded systems especially, with all the GCC merits coming with it. Even though the latest JDK implementations are not yet completed, it is being improved constantly.

9.5 Microsoft C# Compiler for .NET Framework

9.5.1 Introduction to .NET Framework

The .NET Framework (Microsoft, 2011) is an integral Microsoft Windows component for building and running applications. It is a multi-language development platform that comprises a virtual machine called the *Common Language Runtime* (CLR), and a library called the *Framework Class Library* (FCL) (MSDN Library, 2011).

The CLR is the heart of the .NET framework, and it consists of several features, including a class loader, metadata engine, garbage collector, debugging services and security services. The CLR is Microsoft's implementation for the Common Language Infrastructure (CLI) Standard, which defines an execution environment that allows multiple high-level languages to be used on different computer platforms[9] without being rewritten for specific architectures (CLI Spec, 2006). It takes care of compiling intermediate language programs to native machine code, memory management (e.g. allocation of objects and buffers), thread execution, code execution and exception handling, code safety[10] verification (e.g. array bounds and index checking), cross-language integration (i.e. following certain rules to ensure the interoperability between languages), garbage collection, and controls the interaction with the OS.

FCL is the other core component of .NET Framework, which is a library of classes, interfaces, and value types. It is structured as a hierarchical tree and divided into logical groupings of types called "namespaces" (as in the Java API) according to their functionality. "System" is the root for all types in the .NET Framework's namespace hierarchy. The class libraries provided with the .NET Framework can be accessed using

[9] Such platforms would be the combination of any architecture with any version of the Microsoft Windows operating system running atop.

[10] Comprising type, memory and control-flow safety, meaning that the intermediate code is correctly generated and only accesses the memory locations it is authorized to access, and isolating objects from each other to protect them from accidental or malicious corruption.

any language that targets the CLR. One can combine the object-oriented collection of reusable types and common functions from the FCL with his/her own code, which can be written in any of the .NET compatible languages. The .NET compatible languages can altogether be called ".NET languages". C#, C++, Perl, Python, Ruby, Scheme, Visual Basic, Visual J++, Phalanger[11], and FORTRAN. NET[12] are a few examples to .NET languages with CLR-targeting compilers (Hamilton, 2003).

Programs written in .NET languages are first compiled into a stack-based byte-code format named as *Common Intermediate Language* (CIL)[13]. Then this CIL code is fed into the CLR, with the associated *metadata*. Metadata in .NET is binary information that describes the classes and methods (with their declarations and implementations), data types, references to other types and members, and attributes (e.g. versioning information, security permissions, etc.) of the program. Metadata is stored in a file called the *Manifest*. CIL code and the manifest are wrapped in a Portable Executable[14] file that is called an *Assembly*. An assembly can consist of one or more program files and a manifest, possibly along with reference files, such as bitmap files that are called within the program. Assemblies can be either "Process Assemblies" (EXE files) or "Library Assemblies" (DLL files).

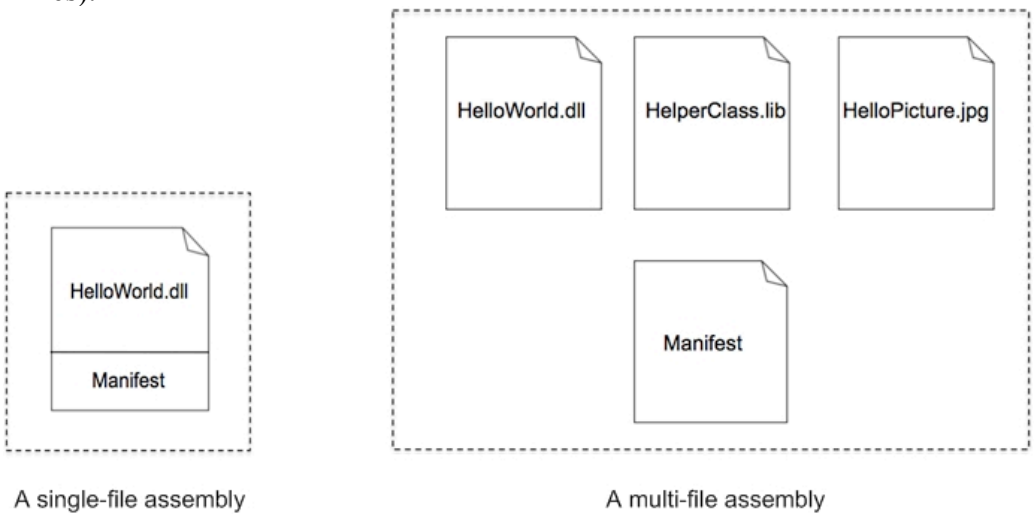

Figure 9.3 Single-file vs. multi-file assembly

CIL code contained in an assembly with metadata is *managed code*. The program code that executes under the management of a VM is called "managed code", whereas the

[11] An implementation of PHP with extensions for ASP.NET.
[12] Fortran compiling to .NET.
[13] Also referred to as "Intermediate Language" (IL), or formerly "Microsoft Intermediate Language" (MSIL).
[14] The Portable Executable (PE) format is a file format for executables, object code, and DLLs, used in 32-bit and 64-bit versions of Windows operating systems.

program code that is directly compiled into machine code before execution and executed directly by the CPU is called “unmanaged code”. Managed code is designed to be more reliable and robust than unmanaged code, with features such as garbage-collection and type-safety available to it. The managed code running in the CLR cannot be accessed outside the runtime environment and or cannot call Operating System (OS) services directly from outside the runtime environment. This makes programs more isolated and computers more secure. Managed code needs the CLR’s JIT Compiler to convert it to native executable code, and it relies on CLR to provide services such as security, memory management and threading during runtime.

On the other hand, pre-compiled executables are called *unmanaged code*. One may prefer bypassing CLR and make direct calls to specific OS services through the Win32 API. However, such code would be considered unsafe since it yields security risks, and it will throw an exception if fed to CLR since it is not verifiable. Unmanaged code is directly compiled to architecture specific machine code. Hence, unmanaged code is not portable.

Figure 9.4 depicts the compilation of programs written in .NET languages for the .NET Framework. A program is first compiled into an executable file (i.e. assembly) by the individual higher-level language compiler, and then the executable file is fed into CLR to be JIT-compiled into native code to be executed (MSDN Library, 2011).

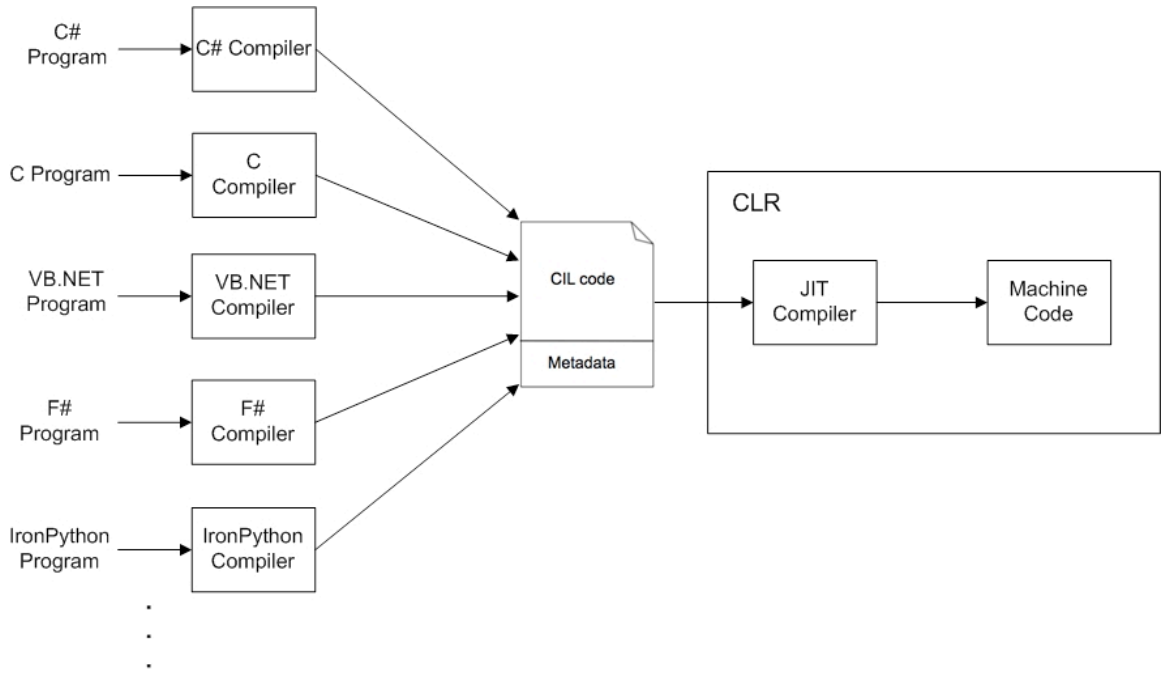

Figure 9.4 Language Integration in .NET Framework

A significant advantage of the .NET Framework is cross-language interoperability, which is supported on all versions of Windows OS, regardless of the underlying architecture. Language interoperability means that code written in one language can interact with code

written in another language. This is made possible in .NET languages and language-specific compilers by following certain set of rules comprising the *Common Type System* (CTS).

CTS is a unified type system that supports the types found in many programming languages; and is shared by all .NET language-specific compilers and the CLR. It is the model that defines the rules that the CLR follows when declaring, using, and managing runtime types. The CTS also defines casting rules (including boxing and un-boxing operations), scopes, and assemblies. CTS supports object-oriented languages, functional languages and procedural programming languages. By following the rules of CTS, programmers can mix constructs from different .NET languages, such as passing an instance of a class to a method of a class written in a different programming language, or defining a type in one language and deriving from that type when creating a new type in another language. The types defined in CTS can be categorized as the *Value types* and the *Reference Types*.

Value types are stored in the stack rather than the garbage-collected heap. Each value type describes the storage that it occupies, the meanings of the bits in its representation, and the valid operations on that representation. Value types can be either *built-in data types* or *user-defined types* such as Enum.

Reference types are passed by reference and stored in the heap. A Reference type carries more information than a Value type. It has an identity that distinguishes it from all other objects, and it has slots that store other entities, which can be either objects or values. Reference types can be categorized as *self-describing reference types*, *built-in reference types*, *interfaces* and *pointers*. Self-describing types can be further categorized as arrays and class types. The class types are user-defined classes, boxed value types, and delegates (the managed alternative to unmanaged function pointers). Built-in reference types include *Object* (primary base class of all classes in the .NET Framework, which is the root of the type hierarchy) and *String*.

Of course, a disadvantage of .NET is that it runs only on Windows platforms. One doesn't get the "write once, run anywhere" flexibility of Java.

9.5.2 Microsoft C# Compiler

When a C# program is fed to Microsoft's C# compiler, the compiler generates metadata about the program and the corresponding CIL code. In order to achieve this, the compiler does multiple passes over the code, which can be grouped into two main passes followed by many more sub-passes.

In the first main pass, it looks for declarations in the code to gather information about the used namespaces, classes, structs, enums, interfaces, delegates, methods, type parameters, formal parameters, constructors, events and attributes. All this information is extracted over more sub-passes. A lexical analyzer identifies the tokens in the source file, and the

parser does a top-level parse, not going inside method bodies. Both the lexical analyzer and parser are hand-written, the parser being a basic recursive-descent parser. Then, a "declaration" pass records the locations of name spaces and type declarations in the program.

After the declaration pass, the compiler does not need to go over the actual program code; it can do the next passes on the symbols it has generated so far to extract further metadata. The next pass verifies that there are no cycles in the base types of the declared types. A similar pass is also done for the generic parameter constraints on generic types, verifying the acyclic hierarchy. After that, another pass checks the members (methods of classes, fields of structs and enum values, etc.) of every type. For example, whether what overriding methods override are actually virtual methods, or enums have no cycles, etc. One last pass takes care of the values of all const fields.

In the second main pass the compiler generates CIL code. This time the compiler parses the method bodies and determines the type of every expression within the method body. Many more sub-passes follow this. As in the first main pass, the compiler does not need the actual program code once it has created the annotated parse tree; subsequent passes work on this data structure, re-writing it as necessary.

First, the compiler transforms loops into gotos and labels. Then, more passes are done to look for problems and to do some optimization. The compiler makes a pass for each of the following:

- to search for use of deprecated types and generate warnings if any exist,
- to search for types that have no metadata yet and emit those,
- to check whether expression trees[15] are constructed correctly,
- to search for local variables that are defined but not used, and generate warnings,
- to search for illegal patterns inside iterator blocks,
- to search for unreachable code, such as a non-void method with no return statement, and generate warnings,
- to check whether every goto targets a sensible label, and every label is targeted by a reachable goto, and
- to check whether all local variables have assigned values before their use.

Once the compiler is done with the passes for problem checks, it initiates the set of optimizing passes:

- to transform expression trees into the sequence of factory method calls necessary to create the expression trees at runtime,

[15] Expression trees are C#-specific tree-like data structures, where each node is an expression in the code, for example, a method call or a binary operation such as x < y. For details on expression trees, see (MSDN Library, 2010).

- to rewrite all arithmetic that can possibly have null value as code that checks for null values,
- to rewrite references to methods defined in base classes as non-virtual calls,
- to find unreachable code and remove it from the tree, since generating IL for such code is redundant,
- to rewrite `switch(constant)` expressions as a direct branch to the correct case,
- to optimize arithmetic, and
- to transform iterator blocks into switch-based state machines.

After all these passes[16], the compiler can finally generate IL code by using the latest version of the annotated parse tree (Lippert, 2010). The Microsoft C# compiler is written in unmanaged C++, and generates IL code structures as a sequence of basic blocks. The compiler can apply further optimizations on the basic blocks; for instance, it can rewrite branches between basic blocks for a more efficient call sequence, or remove the basic blocks containing dead code. Generated metadata and IR code are then fed to the CLR as an executable, to be JIT-compiled on any architecture running a Windows OS.

9.5.3 Classic Just-in-time Compilation in the CLR

Microsoft's .NET runtime has two JIT compilers: the *Standard-JIT compiler* and the *Econo-JIT compiler*. The Econo-JIT compiler works faster than the Standard-JIT compiler. It compiles only Common Intermediate Language (CIL) code for methods that are invoked during runtime, but the native code is not saved for further calls. The Standard-JIT generates more optimized code than does the Econo-JIT and verifies the CIL code; compiled code is stored in cache for subsequent invocations.

When an application is executed, Windows OS checks whether it is a .NET assembly; if so, the OS starts up the CLR and passes the application to it for execution. The first thing CLR's JIT compiler does is to subject the code to a verification process to determine whether the code is type-safe by examining the CIL code and the associated metadata. Besides checking that the assembly complies with the CTS, CLR's JIT compiler uses the metadata to locate and load classes, discover information about the program's classes, members and inheritance, lay out instances in memory, and resolve method invocations during runtime. The executed assembly may refer to other assemblies; in this case the referenced assemblies are loaded as needed.

[16] There are some more passes for .NET-specific tasks dealing with COM objects, anonymous types and functions, and dynamic calls using DLR; however, they are excluded here to avoid introducing extra terminology and unnecessary detail.

When a type (e.g. a structure, interface, enumerated type or primitive type) is loaded, the loader creates and attaches a stub[17] to each of the type's methods. All types have their own *method table* to store the addresses of their stubs. All object instances of the same type will refer to the same method table. Each method table contains information about the type, such as whether it is an interface, abstract class or concrete class; the number of interfaces implemented by the type; the interface map for method dispatch; the number of slots in the method table; and an embedded method slot table, which points to the method implementations called *method descriptors*. Method slot table entries are always ordered as inherited virtuals, introduced virtuals, instance Methods, and static methods. As Figure 9.5 shows, *ToString*, *Equals*, *GetHashCode*, and *Finalize* are the first four methods in the method slot table for any type. These are virtual methods inherited from **`System.Object`**. *.cctor* and *.ctor* are grouped with static methods and instance methods, respectively.

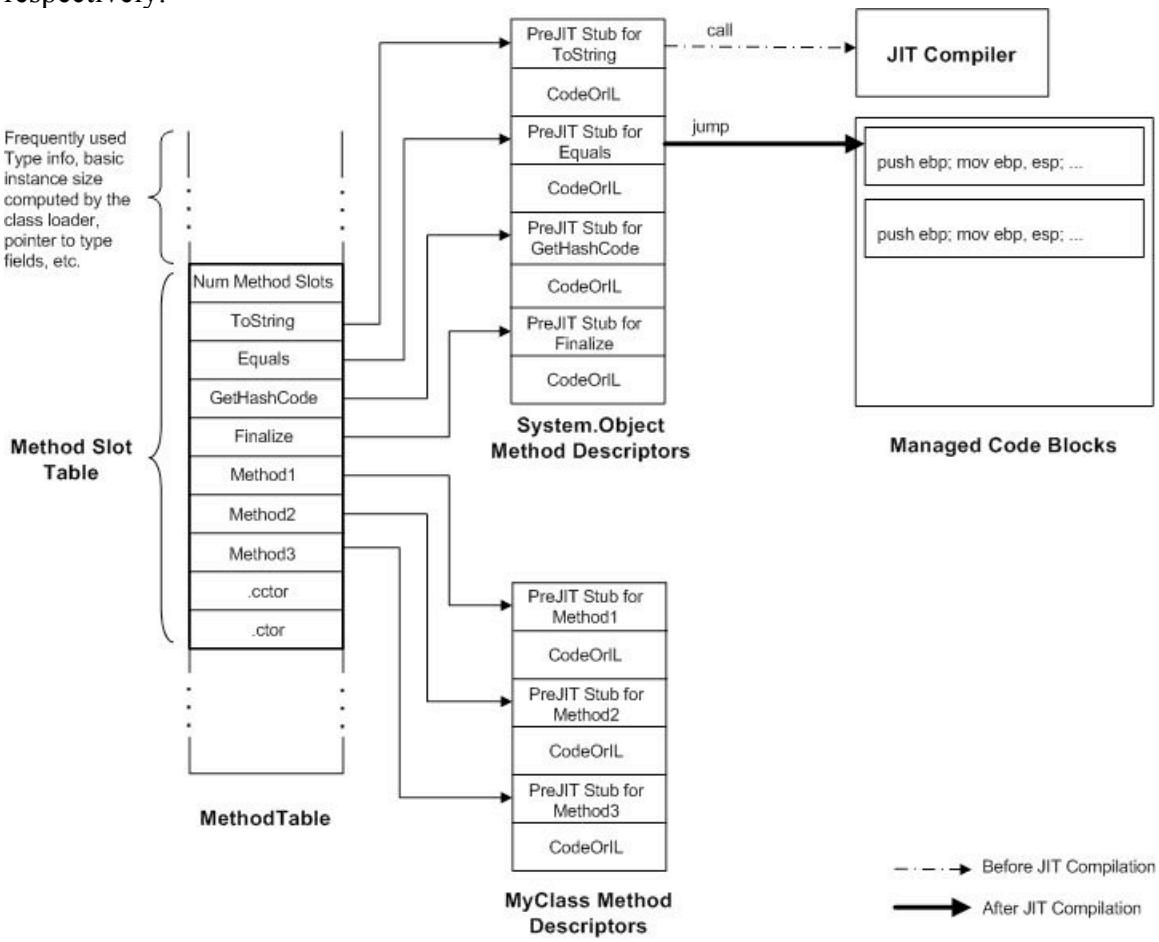

Figure 9.5 The Method Table in .NET

[17] A Stub is a routine that only declares itself and the parameters it accepts and returns an expected value for the caller. A stub contains just enough code to allow it to be compiled and linked with the rest of the program; it does not contain the actual complete implementation.

Initially, all entries of all method tables refer to the JIT compiler through method descriptors. Method descriptors are generated during loading process and they all point to the CIL code (*CodeOrIL* in Figure 9.5) with a padding of 5 bytes (*PreJit Stub* in Figure 9.5) containing the instructions to make a call to the JIT Compiler. As discussed in section 9.2, it is possible that some code might never be called during execution. So instead of converting all of the CIL code into native machine code, the JIT compiler converts the CIL as needed. Once a method is called for the first time, the call instruction contained in the associated stub passes control to the JIT compiler, which converts the CIL code for that method into native code and modifies the stub to direct execution to the location of the native code by overwriting the call instruction with an unconditional jump to the JIT-compiled native code. In other words, while *methods are being stubbed out*, the JIT compiler overwrites the call instruction, replacing the address in the method table with the compiled code's memory address. The machine code produced for the stubbed out methods are stored in memory. Subsequent calls to the JIT-compiled method directly invoke the machine code that was generated after the first call, thus, time to compile and run the program is reduced (Kommalapati and Christian, 2005).

Here, when we say "The JIT Compiler converts the CIL code into native code" do not think that it is just a plain conversion. On the contrary, it follows several stages to do its job, like taking care of verification, optimization and memory management. The first stage is *importing*, in which the CIL code is first converted to JIT's internal representation. While doing that, JIT compiler has to make sure that the CIL code is type-safe before converting it into machine code. That means confirming that the code can access memory locations and call methods only through properly defined types. When a piece of code is verifiably type safe, it means that a reference to a type is strictly compatible with the type being referenced, and only appropriately defined operations are invoked on an object. In this stage, the JIT compiler must closely communicate with the CLR, for example, when it encounters an **`ldfld`**[18] instruction. In such cases, the JIT compiler has to make sure that the field is loaded by CLR, and has to learn whether it can access the field directly, or via some helper.

The second stage is *morphing*, where some transformations are applied to the internal structures to simplify or lightly optimize the code. The examples to such transformations would be constant propagation and method inlining.

[18] The **`ldfld`** is an CIL instruction that pushes the value of a field located in an object onto the stack. The object must be on the stack as an object reference (type O), a managed pointer (type &), an unmanaged pointer (type native int), a transient pointer (type *), or an instance of a value type. The use of an unmanaged pointer is not permitted in verifiable code.

In the next stage, the JIT compiler performs a traditional flow-graph analysis to determine the liveness of variables, loop detection, etc. The information obtained in this stage is used in the subsequent stages.

The fourth stage is about heavy optimizations like common sub-expression and range check elimination, loop hoisting, and so on.

Then comes the register allocation stage, where JIT compiler must effectively map variables to registers.

Finally, the code generation and emitting stages come (Notario, 2004). While generating the machine code, the JIT compiler must consider what processor it is generating code for and the OS version. For example, it may embed the address of some Win32 APIs in to the native code produced, and the address of these APIs could change between different service packs of a specific Windows OS (Richter, 2005). After generating the machine code for a specific method, the JIT compiler packs everything together and returns to the CLR, which will then redirect control to the newly generated code. Garbage collection and debugger information is also recorded here. Whenever memory is low, the JIT compiler will free up memory by placing back the stubs of methods that had not been called frequently during program operation up to that point.

Knowing that JIT compilation in the CLR works on a method-by-method basis as needed, gives us an understanding of the performance characteristics and the ability to make better design decisions. For example, if one knows that a piece of code will be needed only in rare or specific cases, she can keep it in a separate assembly. Alternatively, one can keep that rarely needed piece of code in a separate method so that JIT Compiler will not compile it until explicitly invoked.

One important point to emphasize about the CLR's JIT compilation is that the CLR does not provide an interpreter to interpret the CIL Code. A method's CIL code is compiled to machine code at once to run. In section 9.2 we saw that things work differently in Java's HotSpot JIT compiler. The HotSpot compiler may not JIT compile a method if it expects the overhead of compilation to be lower than the overhead of interpreting the code. It can recompile with heavier optimization than before based on actual usage. In that sense, CLR's JIT compilation is simpler than HotSpot compilation.

At the end one may ask how Java platform compares to .NET Framework. The answer to that, in a nutshell, is that Java platform deals with one language running on multiple operating systems, whereas .NET framework deals with multiple languages on a single (Windows) operating system.

Further Reading

An extensive collection of resources related to HotSpot Technology can be found at Oracle's website (Oracle, 2011) including various publications on HotSpot, FAQ and official technology documentation. Especially (Oracle Arch., 2010) provides comprehensive explanations of the Java HotSpot VM and its compiler.

(Eclipse, 2004) introduces incremental building in Eclipse. (Clayberg and Rubel, 2008) gives a good introduction to plug-in development for Eclipse. (D'Anjou et al, 2004) is a good source to learn about Java development in Eclipse.

(Bothner, 2003) explains GCJ in depth. Also, (GNU, 2010) gives a full list of optimization options available in GCC.

(Richter, 2002) and (Richter, 2010) are both excellent and very different introductions to Microsoft's .NET and the CLR (Anything Jeffrey Richter writes is excellent.) (Hamilton, 2003) presents concepts revolving around language interoperability in the CLR in detail. (Kommalapati and Christian, 2005) examines the internals of the CLR.

(Aycock, 2003) presents the ideas underlying JIT compilation and how it is used in implementing various programming languages.