Chapter 9

Real-world abstract machines

This chapter discusses some widely used real-world *abstract machines*.

9.1 What files are provided for this chapter

File	Contents
`virtual/ex6java.java`	a linked list class in Java; see Figure 9.4
`virtual/ex13.java`	a version of `ex13.c` in Java
`virtual/ex13.cs`	a version of `ex13.c` in C#; see Figure 9.8
`virtual/CircularQueue.cs`	a generic circular queue in C#; see Figure 9.10
`virtual/Selsort.java`	selection sort in Java
`virtual/Selsort.cs`	selection sort in C#

9.2 An overview of abstract machines

An abstract machine is a device, which may be implemented in software or in hardware, for executing programs in an intermediate instruction-oriented language. The intermediate language is often called bytecode, because the instruction codes are short and simple compared to the instruction set of 'real' machines such as the x86, PowerPC or ARM architectures. Abstract machines are also known as *virtual machines*. It is common to identify an machine with the source language it implements, although this is slightly misleading. Prime examples are Postscript (used in millions of printers and typesetters), P-code

(widely used in the late 1970'es in the UCSD implementation of Pascal for microcomputers), the Java Virtual Machine, and Microsoft's Common Language Infrastructure. Many projects exist whose goal is to develop new abstract machines, either to be more general, or for some specific purpose.

The purpose of an abstract machine typically is to increase the portability and safety of programs in the source language, such as Java. By compiling Java to a single bytecode language (the JVM), one needs only a single Java compiler, yet the Java programs can be run with no changes on different 'real' machine architectures and operating systems. Traditionally it was cumbersome to develop portable software in C, say, because an `int` value might have 16, 32, 36 or 64 bit depending on which machine the program was compiled for.

The Java Virtual Machine (JVM) is an abstract machine and a set of standard libraries developed by Sun Microsystems since 1994 [85]. Java programs are compiled to JVM bytecode to make Java programs portable across platforms. There are Java Virtual Machine implementations for a wide range of platforms, from large high-speed servers and desktop computers (Sun's Hotspot JVM, IBM's J9 JVM, and others) to very compact embedded systems (Sun's KVM, Myriad's Jbed, and others). There are even implementations in hardware, such as the AJ-80 and AJ-100 Java processors from aJile Systems [5].

The Common Language Infrastructure is an abstract machine and a set of standard libraries developed by Microsoft since 1999, with very much the same goals as Sun's JVM. The whole platform has been standardized as by Ecma International [41]. Microsoft's implementation of CLI is known as the Common Language Runtime (CLR) and is part of .NET, a large set of tools, libraries and technologies. The first version of CLI was releases in January 2002, and version 2.0 with generics was released in 2005. The subsequent versions 3.5 (2008) and 4.0 (2010) mostly contain changes to the libraries and the source languages (C#, VB.NET), whereas the abstract machine bytecode remains the same as version 2.0.

Whereas JVM was planned as an intermediate target language only for Java, CLI is intended as a target language for a variety of high-level source languages, primarily C#, VB.NET (a successor of Visual Basic) and JScript (a version of Javascript), but also C++, COBOL, Haskell, Standard ML, Eiffel, and more recently F#. In particular, programs written in any of these languages are supposed to be able to interoperate, using the common object model supported by the CLI. This has influenced the design of CLI, whose bytecode language is somewhat more general than that of the JVM, although it is still visibly slanted towards class-based, single-inheritance object-oriented languages such as Java and C#. Also, CLI was designed with just-in-time compilation in mind. For this reason, CLI bytecode instructions are not explicitly typed; the just-in-time compilation phase must infer the types anyway, so there

is no need to give them explicitly.

While the JVM has been implemented on a large number of platforms (Solaris, Linux, MS Windows, web browsers, mobile phones, personal digital assistants) from the beginning, CLI was primarily intended for MS Windows NT/2000/XP/Vista/7 and their successors, and for Windows Compact Edition (CE). However, the Mono project, sponsored by Novell, [95] has created an open source implementation of CLI for many platforms, including Linux, MacOS, Windows, Apple's iPhone, Google's Android phone, and more.

The Parallel Virtual Machine (PVM) is a different kind of virtual machine: it is a library for C, C++ and Fortran programs that makes a network of computers look like a single (huge) computer [105]. Program tasks can easily communicate with each other, even between different processor architectures (x86, Sun Sparc, PowerPC, ...) and different operating systems (Linux, MS Windows, Solaris, HP-UX, AIX, MacOS X, ...). The purpose is to support distributed scientific computing.

9.3 The Java Virtual Machine (JVM)

9.3.1 The JVM runtime state

In general, a JVM runs one or more threads concurrently, but here we shall consider only a single thread of execution. The state of a JVM thread has the following components:

- classes that contain methods, where methods contain bytecode;
- a heap that stores objects and arrays;
- a frame stack;
- class loaders, security managers and other components that we do not care about here.

The *heap* is used for storing values that are created dynamically and whose lifetimes are hard to predict. In particular, all arrays and objects (including strings) are stored on the heap. The heap is managed by a *garbage collector*, which makes sure that unused values are thrown away so that the memory they occupy can be reused for new arrays and objects. Chapter 10 discusses the heap and garbage collection in more detail.

The JVM *frame stack* is a stack of frames (also called activation records), containing one frame for each method call that has not yet completed. For instance, when method `main` has called method `fac` on the argument 3, which

has called itself recursively on the argument 2, and so on, the frame stack has the form shown in Figure 9.1. Thus the stack has exactly the same shape as in the micro-C abstract machine, see Figure 8.3.

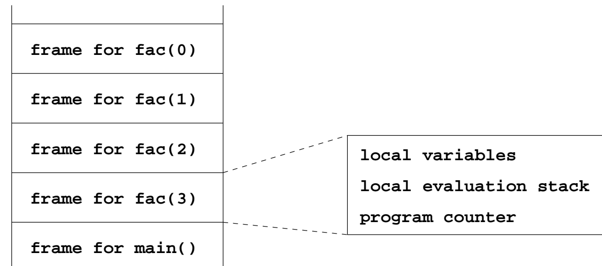

Figure 9.1: JVM frame stack (left) and layout of a stack frame (right).

Each JVM stack frame has at least the following components:

- local variables for this method;
- the local evaluation stack for this method;
- the program counter (pc) for this method.

The local variables include the method's parameters, and also the current object reference (`this`) if the method is non-static. The `this` reference (if any) is the first local variable, followed by the method's parameters and the method's local variables. In the JVM bytecode, a local variable is named by its index; this is essentially the local variable's declaration number. For instance, in a non-static method, the current object reference (`this`) has local variable index 0, the first method parameter has index 1, and so on. In an static method, the first method parameter has index 0, and so on.

In the JVM the size of a value is one 32-bit word (for booleans, bytes, characters, shorts, integers, floats, references to array or object), or two words (longs and doubles). A local variable holding a value of the latter kind occupies two local variable indexes.

Only primitive type values (`int`, `char`, `boolean`, `double`, and so on) and references can be stored in a local variable or in the local evaluation stack. All objects and arrays are stored in the heap, but a local variable and the local evaluation stack can of course hold a reference to a heap-allocated object or array.

As shown in Figure 9.1, and unlike the abstract machine of Chapter 8, the JVM keeps the expression evaluation stack separate from the local variables,

and also keeps the frames of different method invocations separate from each other. All stack frames for a given method must have the same fixed size: the number of local variables and the maximal depth of the local evaluation stack must be determined in advance by the Java compiler.

The instructions of a method can operate on:

- the local variables (load variable, store variable) and the local evaluation stack (duplicate, swap);
- static fields of classes, given a class name and a field name;
- non-static fields of objects, given an object reference and a field name;
- the elements of arrays, given an array reference and an index.

Classes (with their static fields), objects (with their non-static fields), strings, and arrays are stored in the heap.

9.3.2 The Java Virtual Machine (JVM) bytecode

As can be seen, the JVM is a stack-based machine quite similar to the micro-C abstract machine studied in Chapter 8. There is a large number of JVM bytecode instructions, many of which have variants for each argument type. An instruction name prefix indicates the argument type; see Figure 9.2. For instance, addition of integers is done by instruction `iadd`, and addition of single-precision floating-point numbers is done by `fadd`.

Prefix	**Type**
`i`	`int`, `short`, `char`, `byte`
`b`	`byte` (in array instructions only)
`c`	`char` (in array instructions only)
`s`	`short` (in array instructions only)
`f`	`float`
`d`	`double`
`a`	reference to array or object

Figure 9.2: JVM instruction type prefixes.

The main categories of JVM instructions are shown in Figure 9.3 along with the corresponding instructions in Microsoft's CLI.

The JVM bytecode instructions have symbolic names as indicated above, and they have fixed numeric codes that are used in JVM class files. A class file

Category	JVM	CLI
push constant	bipush, sipush, iconst, ldc, aconst_null, ...	ldc.i4, ldc.i8, ldnull, ldstr, ldtoken
arithmetic	iadd, isub, imul, idiv, irem, ineg, iinc, fadd, fsub, ...	add, sub, mul, div, rem, neg
checked arithmetic		add.ovf, add.ovf.un, sub.ovf, ...
bit manipulation	iand, ior, ixor, ishl, ishr, ...	and, not, or, xor, shl, shr, shr.un
compare values		ceq, cgt, cgt.un, clt, clt.un
type conversion	i2b, i2c, i2s, i2f, f2i, ...	conv.i1, conv.i2, conv.r4, ...
load local var.	iload, aload, fload, ...	ldloc, ldarg
store local var.	istore, astore, fstore, ...	
load array element	iaload, baload, aaload, faload, ...	ldelem.i1, ldelem.i2, ldelem.r4, ...
store array element	iastore, bastore, aastore, fastore, ...	stelem.i1, stelem.i2, stelem.r4, ...
load indirect		ldind.i1, ldind.i2, ...
store indirect		stind.i1, stind.i2, ...
load address		ldloca, ldarga, ldelema, ldflda, ldsflda
stack	swap, pop, dup, dup_x1, ...	pop, dup
allocate array	newarray, anewarray, multianewarray, ...	newarr
load field	getfield, getstatic	ldfld, ldstfld
store field	putfield, putstatic	stfld, stsfld
method call	invokevirtual, invokestatic, invokespecial, ...	call, calli, callvirt
load method pointer		ldftn, ldvirtftn
method return	return, ireturn, areturn, ...	ret
jump	goto	br
compare to 0 and jump	ifeq, ifne, iflt, ifle, ifgt, ifge	brfalse, brtrue
compare values and jump	if_icmpeq, if_icmpne, ...	beq, bge, bge.un, bgt, bgt.un, ble, ble.un, blt, blt.un, bne.un
switch	lookupswitch, tableswitch	switch
object-related	new, instanceof, checkcast	newobj, isinst, castclass
exceptions	athrow	throw, rethrow
threads	monitorenter, monitorexit	
try-catch-finally	jsr, ret	endfilter, endfinally, leave
value types		box, unbox, cpobj, initobj, ldobj, stobj, sizeof

Figure 9.3: Bytecode instructions in JVM and CLI.

represents a Java class or interface, containing static and non-static field declarations, and static and non-static method declarations. A JVM reads one or more class files and executes the `public static void main(String[])` method in a designated class.

9.3.3 Java Virtual Machine (JVM) class files

When a Java program is compiled with a Java compiler such as `javac` or `jikes`, one or more class files are produced. A class file `C.class` describes a single class or interface `C`. Nested classes within `C` are stored in separate class files named `C$A`, `C$1`, and so on.

The structure of a somewhat abstracted class file is described by the Standard ML type `class_decl` in file `jvm/Classdecl.sml` in Peter Bertelsen's SML-JVM toolkit [15]. Java-based tools for working with JVM class files include BCEL [21], `gnu.bytecode` [48], Javassist [24, 60], and JMangler [61]. In Chapter 14 we show how to use the former two for runtime-code generation.

Figure 9.4 outlines a Java class declaration `LinkedList`, and the corresponding class file is shown schematically in Figure 9.5.

The main components of a JVM class file are:

- the name and package of the class;
- the superclass, superinterfaces, and access flags (`public` and so on) of the class;
- the constant pool, which contains field descriptions and method descriptions, string constants, large integer constants, and so on;
- the static and non-static field declarations of the class;
- the method declarations of the class, and possibly special methods named `<init>` corresponding to the constructors of the class, and a special methods named `<clinit>` corresponding to a static initializer block in the class;
- the attributes (such as source file name).

For each field declaration (type `field_decl`), the class file describes:

- the name of the field;
- the type of the field;
- the modifiers (`static`, `public`, `final`, ...);
- the attributes (such as source file line number).

```
class LinkedList extends Object {
  Node first, last;

  void addLast(int item) {
    Node node = new Node();
    node.item = item;
    ...
  }

  void printForwards() { ... }
  void printBackwards() { ... }
}
```

Figure 9.4: Java source code for class `LinkedList` (file `virtual/ex6java.java`).

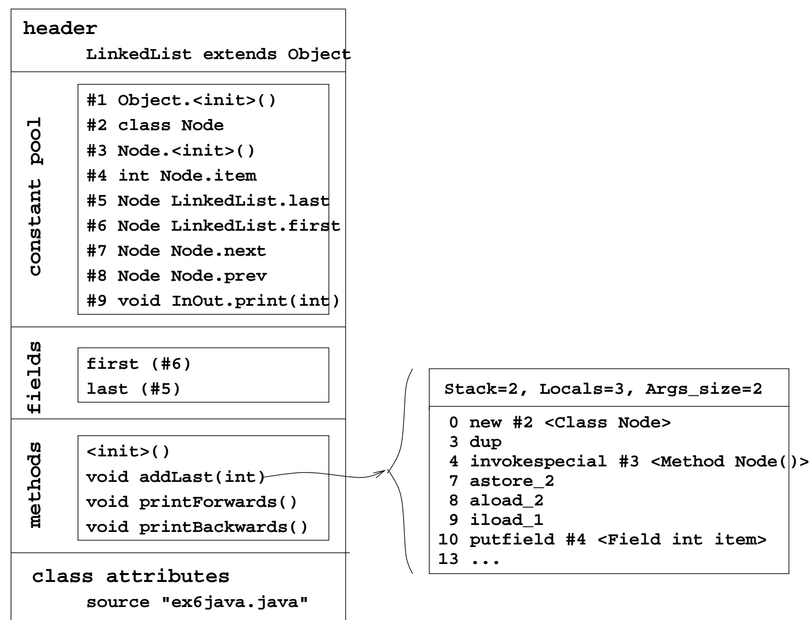

Figure 9.5: JVM class file for class `LinkedList` in Figure 9.4.

For each method declaration (type `method_decl`), the class file describes:

- the name of the method;
- the signature of the method;
- the modifiers (`static`, `public`, `final`, ...);
- the attributes, including
  - the code for the method;
  - the exceptions thrown by the method (from the method's `throws` clause in Java).

The code for a method (attribute `CODE`) includes:

- the maximal depth of the local evaluation stack in the stack frame for the method — this helps the JVM allocate a stack frame of the right size for a method call;
- the number of local variables in the method;
- the bytecode itself, as a list of JVM instructions;
- the exception handlers, that is, `try-catch` blocks, of the method body; each handler (type `exn_hdl`) describes the bytecode range covered by the handler, that is, the `try` block, the entry of the handler, that is, the `catch` block, and the exception class handled by this handler;
- code attributes, such as source file line numbers (for runtime error reports).

To study the contents of a class file `C.class`, whether generated by a Java compiler or the micro-C compiler, you can disassemble it by executing:

```
javap -c C
```

To display also the size of the local evaluation stack and the number of local variables, execute:

```
javap -c -verbose C
```

9.3.4 Bytecode verification

Before a Java Virtual Machine (JVM) executes some bytecode, it will perform so-called *bytecode verification*, a kind of loadtime check. The overall goal is to improve security: the bytecode program should not be allowed to crash the JVM or to perform illegal operations. This is especially important when executing 'foreign' programs, such as applets within a browser, or other downloaded programs or plugins.

Bytecode verification checks the following things, and others, before the code is executed:

- that all bytecode instructions work on stack operands and local variables of the right type;
- that a method uses no more local variables than it claims to;
- that a method uses no more local stack positions than it claims to;
- that a method throws no more exceptions than it claims to do;
- that for every point in the bytecode, the local stack has a fixed depth at that point (and thus the local stack does not grow without bounds);
- that the execution of a method ends with a return or throw instruction (and does not 'fall off the end of the bytecode');
- that execution does not try to use one half of a two-word value (a long or double) as a one-word value (integer or reference or ...).

This verification procedure has been patented. This is a little strange, since the patented procedure (1) is a standard closure (fixed-point) algorithm, and (2) the published patent does not describe the really tricky point: verification of the so-called local subroutines.

9.4 The Common Language Infrastructure (CLI)

Documentation of Microsoft's Common Language Infrastructure (CLI) and its bytecode, can be found on the Microsoft Developer Network [87]. The documentation is included also with the .Net Framework SDK which can be downloaded from the same place.

The CLI implements a stack-based abstract machine very similar to the JVM, with a heap, a frame stack, the same concept of stack frame, bytecode verification, and so on.

A single CIL stack frame contains the same information as a JVM stack frame (Figure 9.1), and in addition has space for local allocation of structs and arrays; see Figure 9.6.

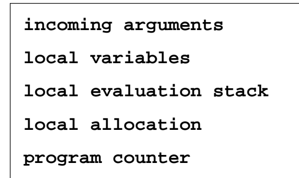

Figure 9.6: A stack frame in Common Language Infrastructure.

CIL is intended as a target language for a range of different source languages, not just Java/C#, and therefore differs from the JVM in the following respects:

- CIL has a more advanced type system than that of JVM, to better support source languages that have parametric polymorphic types (generic types), such as F# and C# 2.0 (see Section 9.5);

- CIL's type system is also more complicated, as it includes several kinds of pointer, native-size integers (that are 32 or 64 bit wide depending on the platform), and so on;

- CIL has support for tail calls (see Section 11.2), to better support functional source languages such as F#, but the runtime system may choose to implement them just like other calls;

- CIL permits the execution of unverified code (an escape from the 'managed execution'), pointer arithmetics etc., to support more anarchic source languages such as C and C++;

- CIL has a canonical textual representation (an assembly language), and there is an assembler `ilasm` and a disassembler `ildasm` for this representation; the JVM has no official assembler format;

- CIL instructions are overloaded on type: there is only one `add` instruction, and load-time type inference determines whether it is an `int` add, `float` add, `double` add, and so on. This reflects a design decision in CIL, to support only just-in-time compilation rather than bytecode interpretation. A just-in-time compiler will need to traverse the bytecode anyway, and can thus infer the type of each instruction instead of just checking it.

When the argument type of a CIL instruction needs to be specified explicitly, a suffix is used; see Figure 9.7. For instance, `ldc.i4` is an instruction for loading 4 byte integer constants.

Suffix	**Type or variant**
`i1`	signed byte
`u1`	unsigned byte
`i2`	signed short (2 bytes)
`u2`	unsigned short or character (2 bytes)
`i4`	signed integer (4 bytes)
`u4`	unsigned integer (4 bytes)
`i8`	signed long (8 bytes)
`u8`	unsigned long (8 bytes)
`r4`	float (32 bit IEEE754 floating-point number)
`r8`	double (64 bit IEEE754 floating-point number)
`i`	natural size signed integer
`u`	natural size unsigned integer, or unmanaged pointer
`r4result`	natural size result for 32-bit floating-point computation
`r8result`	natural size result for 64-bit floating-point computation
`o`	natural size object reference
`&`	natural size managed pointer
`s`	short variant of instruction (small immediate argument)
`un`	unsigned variant of instruction
`ovf`	overflow-detecting variant of instruction

Figure 9.7: CLI instruction types and variants (suffixes).

The main CIL instruction kinds are shown in Figure 9.3 along with the corresponding JVM instructions. In addition, there are some unverifiable (unmanaged) CLI instructions, useful when compiling C or C++ to CLI:

- jump to method (a kind of tail call): `jmp`, `jmpi`
- block memory operations: `cpblk`, `initblk`, `localloc`

The CLI machine does not have the JVM's infamous local subroutines. Instead so-called protected blocks (those covered by `catch` clauses or `finally` clauses) are subject to certain restrictions. One cannot jump out of or return from a protected block; instead a special instruction called `leave` must be executed, causing associated any `finally` blocks to be executed.

A program in C#, F#, VB.Net, and so on, such as the C# program `virtual/ex13.cs` shown in Figure 9.8, is compiled to a CLI file `ex13.exe`.

```
int n = int.Parse(args[0]);
int y;
y = 1889;
while (y < n) {
  y = y + 1;
  if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)
    InOut.PrintI(y);
}
InOut.PrintC(10);
```

Figure 9.8: A source program in C#. The corresponding bytecode is shown in Figure 9.9.

Despite the '.exe' suffix, the resulting file is not a classic MS Windows .exe file, but consists of a small stub that starts the .NET CLI virtual machine, plus the bytecode generated by the C# compiler. Such a file can be disassembled to symbolic CIL code using

```
ildasm /text ex13.exe
```

This reveals the CIL code shown in the middle column of Figure 9.9. It is structurally identical to the JVM code generated by javac for virtual/ex13.java.

9.5 Generic types in CLI and JVM

As can be seen, in many respects the CLI and JVM abstract machines are similar, but their treatment of generic types and generic methods differs considerably. Whereas the CLI supports generic types and generic methods also at the bytecode level (since version 2 from 2005), the JVM bytecode has no notion of generic types or methods. This means that generic types and methods in Java are compiled to JVM bytecode by *erasure*, basically replacing each unconstrained type parameter T as in C<T> by type Object in the bytecode, and replacing each constrained type parameter as in C<T extends Sometype> by its bound Sometype>. The consequences of this are explored in Section 9.5.2 below.

9.5.1 A generic class in bytecode

To illustrate the difference between the CLI's and JVM's implementation of generics, consider the generic circular queue class shown in Figure 9.10.

An excerpt of the CLI bytecode for the circular queue class is shown in Figure 9.11. One can see that class CircularQueue is generic also at the CLI

```
    JVM                      |            CIL                 | Source
-----------------------------+--------------------------------+-----------
  0 aload_0                  |  IL_0000:  ldarg.0             |  args
  1 iconst_0                 |  IL_0001:  ldc.i4.0            |
  2 aaload                   |  IL_0002:  ldelem.ref          |  args[0]
  3 invokestatic #2 (...)    |  IL_0003:  call       (...)    |  parse int
  6 istore_1                 |  IL_0008:  stloc.0             |  n = ...
  7 sipush 1889              |  IL_0009:  ldc.i4     0x761    |
 10 istore_2                 |  IL_000e:  stloc.1             |  y = 1889;
 11 goto 43                  |  IL_000f:  br.s       IL_002f  |  while (...) {
 14 iload_2                  |  IL_0011:  ldloc.1             |
 15 iconst_1                 |  IL_0012:  ldc.i4.1            |
 16 iadd                     |  IL_0013:  add                 |
 17 istore_2                 |  IL_0014:  stloc.1             |  y = y + 1;
 18 iload_2                  |  IL_0015:  ldloc.1             |
 19 iconst_4                 |  IL_0016:  ldc.i4.4            |
 20 irem                     |  IL_0017:  rem                 |
 21 ifne 31                  |  IL_0018:  brtrue.s   IL_0020  |  y % 4 == 0
 24 iload_2                  |  IL_001a:  ldloc.1             |
 25 bipush 100               |  IL_001b:  ldc.i4.s   100      |
 27 irem                     |  IL_001d:  rem                 |
 28 ifne 39                  |  IL_001e:  brtrue.s   IL_0029  |  y % 100 != 0
 31 iload_2                  |  IL_0020:  ldloc.1             |
 32 sipush 400               |  IL_0021:  ldc.i4     0x190    |
 35 irem                     |  IL_0026:  rem                 |
 36 ifne 43                  |  IL_0027:  brtrue.s   IL_002f  |  y % 400 == 0
 39 iload_2                  |  IL_0029:  ldloc.1             |
 40 invokestatic #3 (...)    |  IL_002a:  call       (...)    |  print y
 43 iload_2                  |  IL_002f:  ldloc.1             |
 44 iload_1                  |  IL_0030:  ldloc.0             |
 45 if_icmplt 14             |  IL_0031:  blt.s      IL_0011  |  (y < n) }
 48 bipush 10                |  IL_0033:  ldc.i4.s   10       |
 50 invokestatic #4 (...)    |  IL_0035:  call       (...)    |  newline
 53 return                   |  IL_003a:  ret                 |  return
```

Figure 9.9: Similarity of bytecode generated from Java source and the C# source in Figure 9.8.

```
class CircularQueue<T> {
  private readonly T[] items;
  private int count = 0, deqAt = 0;
  ...
  public CircularQueue(int capacity) {
    this.items = new T[capacity];
  }
  public T Dequeue() {
    if (count > 0) {
      count--;
      T result = items[deqAt];
      items[deqAt] = default(T);
      deqAt = (deqAt+1) % items.Length;
      return result;
    } else
      throw new ApplicationException("Queue empty");
  }
  public void Enqueue(T x) { ... }
}
```

Figure 9.10: A generic class implementing a circular queue, in C#.

bytecode level, taking type parameter `T` which is used in the types of the class's fields and its methods.

Contrast this with Figure 9.12, which shows the JVM bytecode obtained from a Java version of the same circular queue class. There is no type parameter on the class, and the methods have return type and parameter type `Object`, so the class is not generic at the JVM level.

9.5.2 Consequences for Java

The absence of generic types in the JVM bytecode has some interesting consequences for the Java language, not only for the JVM:

- Since type parameters are replaced by type `Object` in the bytecode, a type argument in Java must be a reference type such as `Double`; it cannot be a primitive type such as `double`. This incurs runtime wrapping and unwrapping costs in Java.
- Since type parameters do not exist in the bytecode, in Java one cannot reliably perform a cast `(T)e` to a type parameter, one cannot use a type parameter in an instance test `(e instanceof T)`, and one cannot perform reflection `T.class` on a type parameter.

```
.class private auto ansi beforefieldinit CircularQueue`1<T>
       extends [mscorlib]System.Object
{
  .field private initonly !T[] items
  ...
  .method public hidebysig instance !T  Dequeue() cil managed { ... }
  .method public hidebysig instance void Enqueue(!T x) cil managed { ... }
}
```

Figure 9.11: CLI bytecode, with generic types, for generic class `CircularQueue` in Figure 9.10. The class takes one type parameter, hence the `` `1 `` suffix on the name; the type parameter is called `T`; and the methods have return type and parameter type `T` — in the bytecode, this is written `!T`.

```
class CircularQueue extends java.lang.Object{
  ...
  public java.lang.Object dequeue(); ...
  public void enqueue(java.lang.Object); ...
}
```

Figure 9.12: JVM bytecode, without generic types, for a Java version of generic class `CircularQueue` in Figure 9.10. The class takes no type parameters, and the methods have return type and parameter type `Object`.

- Since a type parameter is replaced by `Object` or another type bound, in Java one cannot overload methods arguments on different instances of a generic type. For instance, this is not allowed:

```
void put(CircularQueue<Double> cqd) { ... }
void put(CircularQueue<Integer> cqd) { ... }
```

  Namely, in the bytecode the parameter type would be just `CircularQueue` in both cases, so the two methods cannot be distinguished.

- Since type parameters do not exist in the bytecode, in Java one cannot create an array whose element type involves a type parameter, as in `arr=new T[capacity]`. The reason is that when the element type of an array is a reference type, then every assignment `arr[i]=o` to an array element must check that the runtime type of `o` is a subtype of the actual element type with which the array was created at runtime; see Section 4.9.1. Since the type parameter does not exist in the bytecode, it cannot be used as actual element type, so this array element assignment check cannot be performed. Therefore it is necessary to forbid the creation of an array instance whose element type involves a generic type parameter. (However, it is harmless to declare a variable of generic array type, as in `T[] arr;` — this does not produce an array instance).

  It follows that the array creation in the constructor in Figure 9.10 would be illegal in Java. A generic circular queue in Java would instead store the queue's elements in an `ArrayList<T>`, which is invariant in its type parameter and therefore does not need the assignment check; see Section 6.6.

9.6 Decompilers for Java and C#

Because of the need to perform load-time checking ('verification', see Section 9.3.4) of the bytecode in JVM and .NET CLI, the compiled bytecode files contain much so-called *metadata*, such as the name of classes and interfaces; the name and type of fields; the name, return type and parameter types of methods; and so on. For this reason, and because the Java and C# compilers generate relatively straightforward bytecode, one can usually *decompile* the bytecode files to obtain source programs (in Java or C#) that are very similar to the originals.

For instance, Figure 9.13 shows the result of decompiling the .NET CLI bytecode in Figure 9.9, using the Reflector tool [113] originally developed by Lutz Roeder's. The resulting C# is very similar to the original source code shown in Figure 9.8.

```
int num = int.Parse(args[0]);
int i = 0x761;
while (i < num) {
  i++;
  if ((((i % 4) == 0) && ((i % 100) != 0)) || ((i % 400) == 0)) {
    InOut.PrintI(i);
  }
}
InOut.PrintC(10);
```

Figure 9.13: The C# code obtained by decompiling the .NET CLI bytecode in the middle column of Figure 9.9.

There exist several decompilers for JVM and Java also, including Atanas Neshkov's DJ decompiler [97]. Decompilers are controversial because they can be used to reverse engineer Java and C# software that is distributed only in 'compiled' bytecode form, so they make it relatively easy to 'steal' algorithms and other intellectual property. To fight this problem, people develop obfuscators, which are tools that transform bytecode files to make it harder to decompile them. For instance, an obfuscator may change the names of fields and methods to keywords such as `while` and `if`, which is legal in the bytecode but illegal in the decompiled programs. One such tool, call Dotfuscator, is included with Visual Studio 2008.

9.7 History and literature

The book by Smith and Nair [120] gives a comprehensive account of abstract machines and their implementation. It covers the JVM kind of virtual machine as well as virtualization of hardware (not discussed here), as used in IBM mainframes and Intel's recent processors. Diehl, Hartel and Sestoft [38] give a more overview over a range of abstract machines.

The authoritative but informal description of the JVM and JVM bytecode is given by Lindholm and Yellin [85]. Cohen [28] and Bertelsen [14] have made two of the many attempts at a more precise formalization of the Java Virtual Machine. A more comprehensive effort which also relates the JVM and Java source code, is by Stärk, Schmid, and Börger [126].

The Microsoft Common Language Infrastructure is described by Gough [51], Lidin [82], and Stutz [125]. Microsoft's CLI specifications and implementations have been standardized by Ecma International [41].