A library is preexisting code you can call.  
A run-time system is a library that makes assumptions about how the compiler works:
- may use tables generated by the compiler
- may examine or manipulate heap or stack layout
- e.g., GC requires help finding root pointers and type descriptors;
  needs compiler to generate write barriers
A virtual machine is a run-time system that provides/specifies everything the program needs, including the instruction set

The Java Virtual Machine (JVM)

Recall how Java works:
- compiler translates Java source to Java byte code (JBC)
- byte code runs on virtual machine
- virtual machine may execute code via interpretation, JIT compilation, or some combination of the two

The JVM’s machine architecture provides:
- all & only Java’s built-in types
  (invokedynamic was added to the JVM for Java 7, to support Java lambdas & dynamic languages)
- type safety
- definite assignment
- garbage collection
- threads
- global constant pool, per-thread stacks, heap, method (code) area
  [ each stack frame contains
    array of locals & formals
    each slot 32 bits wide (longs and doubles take 2)
    can be reused for temporally disjoint data of different types
  expression evaluation stack
  (sized to accommodate largest expression in the method) ]
implicit references (“registers”) for the current program counter, frame, top of operand stack within frame, symbol table info in constant pool

The JVM also defines the format of `.class` files

At start-up, the JVM
loads the given class file (which must have a `main()`) verifies that it satisfies various invariants  
  type safety  
  no operand stack overflow or underflow  
  all references to the constant pool and the locals array are within bounds  
  all constant pool entries are well formed  
  no inheritance from a final class  
  definite assignment  
  (several of these require `data flow analysis`) allocates and initializes static data links to library classes calls `main()` in a single thread

The Java Byte Code instruction set includes
  load-store  
    back and forth between local variable array and operand stack arithmetic  
    all done implicitly on the operand stack type conversion object management  
    new, field and array element access, reflection push, pop, dup, swap branches, switch  
    specify targets as indices in the instruction array of the current method static and virtual method calls  
    specify target symbolically by name (index in constant pool) throw exception monitor enter, exit (wait, notify, and notifyAll are method calls)
public void insert(int v) {
  node n = head;
  while (n.next != null && n.next.val < v) {
    n = n.next;
  }
  if (n.next == null || n.next.val > v) {
    node t = new node();
    t.val = v;
    t.next = n.next;
    n.next = t;
  } // else v already in set
}

Code:
Stack=3, Locals=4, Arge_size=2
0: aload_0       // this
1: getfield      #4; //Field head:LLset$node;
4: astore_2
5: aload_2      // n
6: getfield      #5; //Field LLset$node.next:LLset$node;
9: ifnull       31 // conditional branch
12: aload_2
13: getfield      #5; //Field LLset$node.next:LLset$node;
16: getfield      #6; //Field LLset$node.val:I
19: iload_1      // v
20: if_icmpge    31
23: aload_2
24: getfield      #5; //Field LLset$node.next:LLset$node;
27: astore_2
28: goto        5
31: aload_2
32: getfield      #5; //Field LLset$node.next:LLset$node;
35: ifnull      49
38: aload_2
39: getfield      #5; //Field LLset$node.next:LLset$node;
42: getfield      #6; //Field LLset$node.val:I
45: iload_1
46: if_icmple    76
49: new          #2; //class LLset$node
52: dup
53: aload_0
54: invokespecial #3; //Method LLset$node."<init>":(LLset;)V
57: astore_3
58: aload_3      // t
59: iload_1
60: putfield      #6; //Field LLset$node.val:I
63: aload_3
64: aload_2
65: getfield      #5; //Field LLset$node.next:LLset$node;
68: putfield      #5; //Field LLset$node.next:LLset$node;
71: aload_2
72: aload_3
73: putfield      #5; //Field LLset$node.next:LLset$node;
76: return

Figure 16.2 Java source and bytecode for a list insertion method. Output on the right was produced by Oracle’s javac (compiler) and javap (disassembler) tools, with additional comments inserted by hand.
The Common Language Infrastructure (CLI) is similar to the JVM but more general (The Common Language Runtime [CLR] is Microsoft’s implementation) explicit support for multiple programming languages (Microsoft supports C#, F#, Visual Basic, Managed C++, and JScript) richer common type system (CTS) richer calling mechanisms (including tail recursion) multiple pointer and reference types support for unsafe code etc.

Common Intermediate Language (CIL) is the JBC analogue JIT-centric: several tradeoffs made against interpretation type information in objects, not opcodes separate spaces for arguments and locals built-in support for generics

----------------------------------------

Lazy binding of machine code

JIT

tradeoff between load time and optimization quality
HotSpot has “client” and “server” modes faster than you might think (heavy lifting done by javac) incremental compilation compilation of hot methods in parallel with interpretation caching of machine code across runs dynamic inlining

Binary translation

FX!32 (x86 -> Alpha)
Apple Rosetta (PowerPC -> x86), Rosetta2 (x86 -> ARM)
challenges:
where are the function boundaries?
what are types of data in memory?
what locations are targets of branches?
self-modifying code dynamically generated code introspection
Binary rewriting
  trace scheduling
    HP Dynamo (PA-RISC) & DynamoRIO (x86), late 1990s
instrumentation
  statistics gathering
  simulate new architectures
  insert dynamic semantic checks
  sandboxing (a.k.a. software fault isolation -- SFI)
Pin, Valgrind tools
  trace-based

```prolog
procedure print_matching(S : set, p : predicate)
  foreach e in S
    if p(e)
      print e
```