編譯器設計

Java Virtual Machine

Java Virtual Machine (JVM)

- When you say "Java virtual machine." you may be talking about
  - the abstract specification
    - a concept, described in detail in the book: *The Java Virtual Machine Specification*, 2nd Ed. by Tim Lindholm and Frank Yellin
  - a concrete implementation
    - exist on many platforms and come from many vendors
    - either all software or a combination of hardware and software
  - or a runtime instance
    - hosts a single running Java application
- Each Java application runs inside a runtime instance of some concrete implementation of the abstract specification of the Java virtual machine

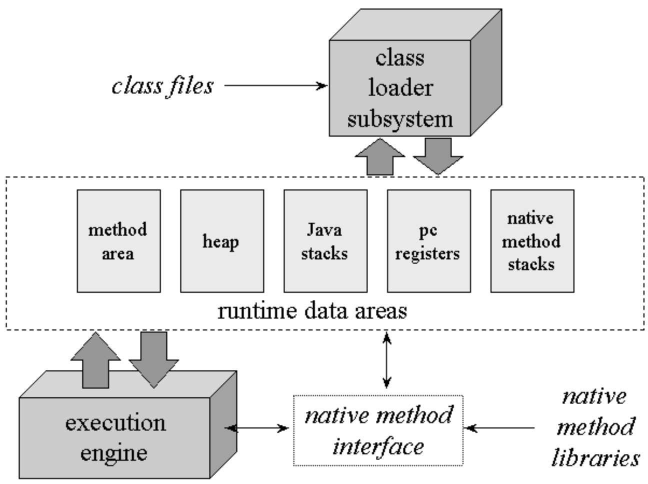
The Life Time of a JVM

- A runtime instance of the Java virtual machine has a clear mission in life
  - to run one Java application
  - When a Java application starts, a runtime instance is born.
  - When the application completes, the instance dies
  - Each Java application runs inside its own Java virtual machine
  - A Java virtual machine instance starts running its solitary application by invoking the main() method of some initial class

```
class Echo {
    public static void main(String[] args) {
        int len = args.length;
        for (int i = 0; i < len; ++i) {
            System.out.print(args[i] + " ");
        }
        System.out.println();
    }
}
```

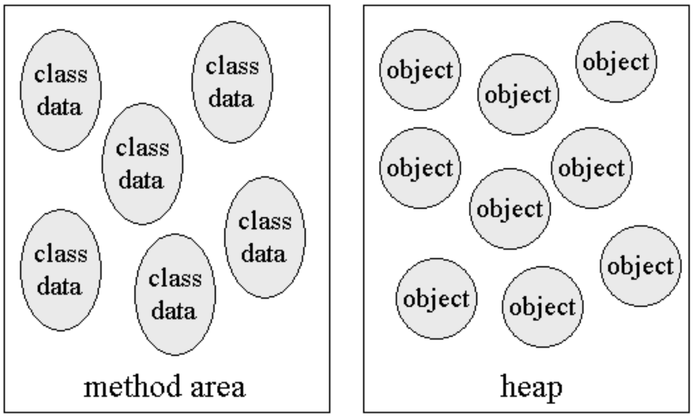

JVM Architecture

- The behavior of a virtual machine instance
  - described in terms of subsystems, memory areas, data types, and instructions in the Java virtual machine specification
  - These components describe an abstract inner architecture for the abstract Java virtual machine
    - to provide a way to strictly define the external behavior of implementations



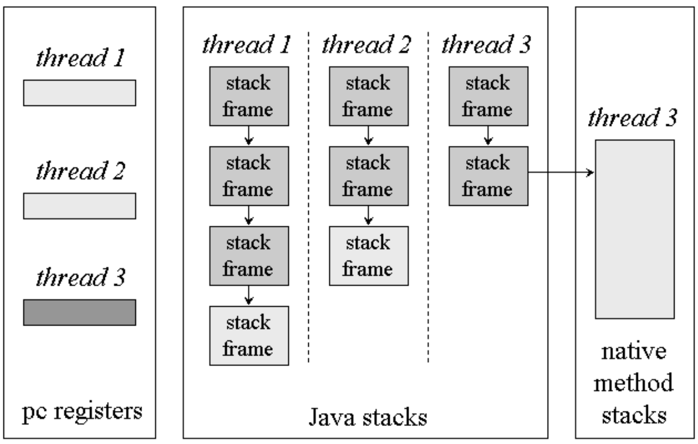
JVM Architecture

- The JVM organizes the memory it needs to execute a program into several *runtime data areas*
  - Each instance of the JVM has one *method area* and one *heap*
    - These areas are shared by all threads running inside the virtual machine



JVM Architecture

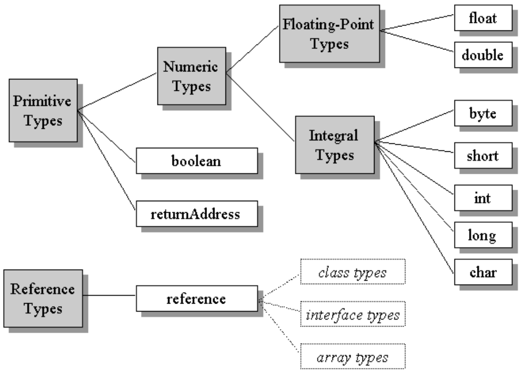
- As each new thread comes into existence, it gets its own *pc register* (program counter) and *Java stack*
  - If the thread is executing a Java method, the value of the pc register indicates the next instruction to execute
  - A thread's Java stack stores the state of Java (not native) method invocations for the thread
    - The Java stack is composed of *stack frames* (or *frames*)



JVM Architecture

◆ Data types

- The data types can be divided into a set of *primitive types* and a *reference type*
  - Reference values refer to objects, but are not objects themselves.
  - Primitive values, by contrast, do not refer to anything. They are the actual data themselves



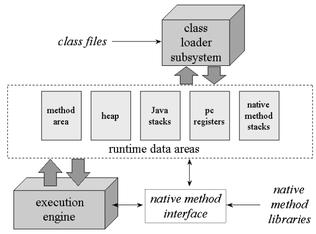
JVM Architecture

◆ Data types

- All the primitive types of the Java programming language are primitive types of the Java virtual machine
  - Although boolean qualifies as a primitive type of the Java virtual machine, the instruction set has very limited support for it
- The Java virtual machine specification defines the range of values for each of the data types, but does not define their sizes

Type	Range
`byte`	8-bit signed two's complement integer (-2^7 to $2^7 - 1$, inclusive)
`short`	16-bit signed two's complement integer (-2^{15} to $2^{15} - 1$, inclusive)
`int`	32-bit signed two's complement integer (-2^{31} to $2^{31} - 1$, inclusive)
`long`	64-bit signed two's complement integer (-2^{63} to $2^{63} - 1$, inclusive)
`char`	16-bit unsigned Unicode character (0 to $2^{16} - 1$, inclusive)
`float`	32-bit IEEE 754 single-precision float
`double`	64-bit IEEE 754 double-precision float
`returnAddress`	address of an opcode within the same method
`reference`	reference to an object on the heap, or `null`

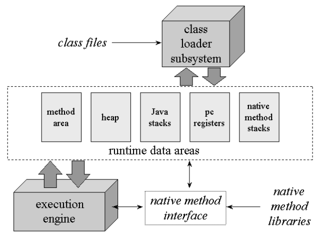
JVM Architecture



◆ Class loader subsystem

- The part of a Java virtual machine implementation that takes care of finding and loading types
- The class loader subsystem is responsible for more than just locating and importing the binary data for classes
  - Loading: finding and importing the binary data for a type
  - Linking: performing verification, preparation, and (optionally) resolution
    - Verification: ensuring the correctness of the imported type
    - Preparation: allocating memory for class variables and initializing the memory to default values
    - Resolution: transforming symbolic references from the type into direct references.
  - Initialization: invoking Java code that initializes class variables to their proper starting values

JVM Architecture



◆ Method Area

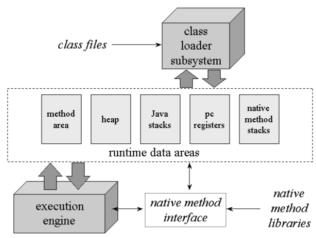
- Information about loaded types is stored in a logical area of memory called the method area
  - The virtual machine extracts information about the type from the binary data and stores the information in the method area
  - Memory for class (static) variables declared in the class is also taken from the method area
- The virtual machine will search through and use the type information stored in the method area as it executes the application it is hosting
  - Designers must attempt to devise data structures that will facilitate speedy execution of the Java application
- All threads share the same method area, so access to the method area's data structures must be designed to be thread-safe

JVM Architecture

◆ Constant pool

- For each type it loads, a Java virtual machine must store a *constant pool*
  - A constant pool is an ordered set of constants used by the type, including literals (string, integer, and floating point constants) and symbolic references to types, fields, and methods
  - Entries in the constant pool are referenced by index
- Constant pool serves a function similar to that of a symbol table for a conventional programming language
- Constant pool plays a central role in the dynamic linking of Java programs
- Each constant pool is allocated from the JVM's method area

JVM Architecture



◆ Heap

- Whenever a class instance or array is created in a running Java application, the memory for the new object is allocated from a single heap
  - As there is only one heap inside a Java virtual machine instance, all threads share it
    - proper synchronization of multi-threaded access to objects (heap data) is needed
  - Because a Java application runs inside its "own" exclusive Java virtual machine instance, there is a separate heap for every individual running application
- The Java virtual machine has an instruction that allocates memory on the heap for a new object, but has no instruction for freeing that memory
  - The virtual machine itself is responsible for deciding whether and when to free memory occupied by objects that are no longer referenced by the running application

JVM Architecture

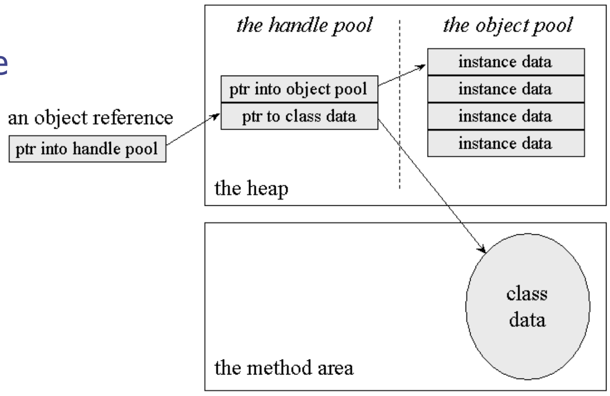
- Garbage collection
  - A garbage collector's primary function is to automatically reclaim the memory used by objects that are no longer referenced by the running application
  - It may also move objects as the application runs to reduce heap fragmentation
  - No garbage collection technique is dictated by the Java virtual machine specification
  - Because references to objects can exist in many places--Java Stacks, the heap, the method area, native method stacks--the choice of garbage collection technique heavily influences the design of an implementation's runtime data areas

JVM Architecture

- Object representation
  - The JVM specification is silent on how objects should be represented on the heap
  - The primary data that must in some way be represented for each object is the instance variables declared in the object's class and all its superclasses
    - Given an object reference, the virtual machine must be able to quickly locate the instance data for the object
    - In addition, there must be some way to access an object's class data (stored in the method area) given a reference to the object
    - For this reason, the memory allocated for an object usually includes some kind of pointer into the method area

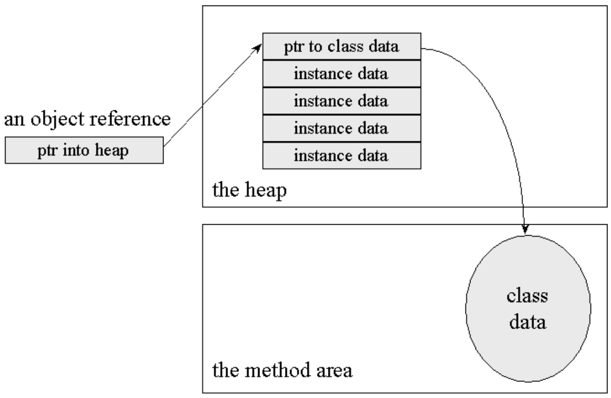
JVM Architecture

- Object representation
  - One possible heap design divides the heap into two parts: a handle pool and an object pool
    - An object reference is a native pointer to a handle pool entry
    - A handle pool entry has two components: a pointer to instance data in the object pool and a pointer to class data in the method area
  - Advantage
    - it makes it easy for the virtual machine to combat heap fragmentation
  - Disadvantage
    - every access to an object's instance data requires dereferencing two pointers



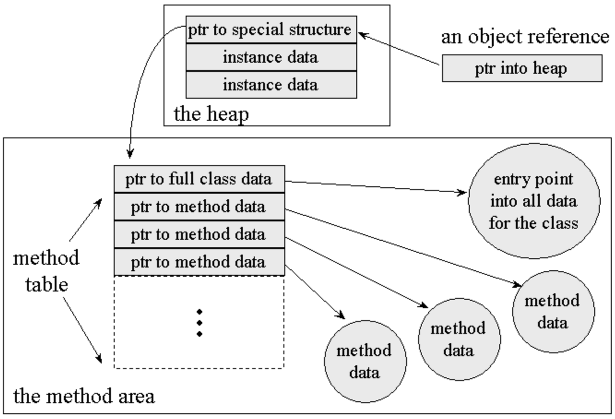
JVM Architecture

- Object representation
  - Another design makes an object reference a native pointer to a bundle of data that contains the object's instance data and a pointer to the object's class data
  - Advantage
    - it requires dereferencing only one pointer to access an object's instance data
  - Disadvantage
    - it makes moving objects more complicated



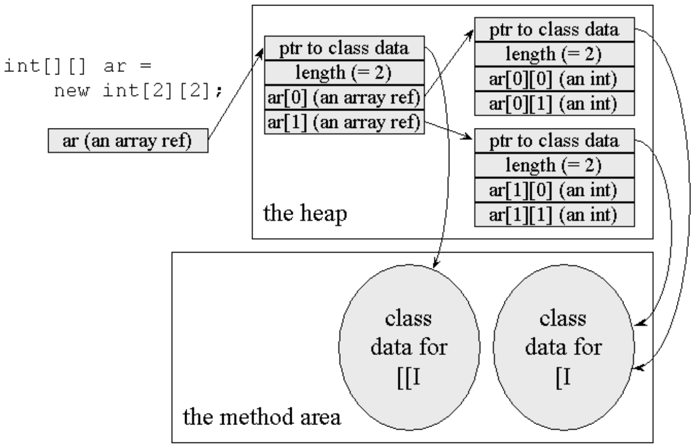
JVM Architecture

- Object representation
  - Maybe a better design is to connect a method table to an object reference
    - The method table is an array of pointers to the data for each instance method that can be invoked on objects of that class
  - Advantage
    - the pointers in the method table may point to methods defined in the object's class or any of its superclasses

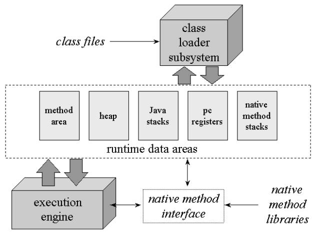


JVM Architecture

- Array representation
  - In Java, arrays are full-fledged objects
    - Like objects, arrays are always stored on the heap
    - Arrays have a Class instance associated with their class, just like any other object
    - Multi-dimensional arrays are represented as arrays of arrays



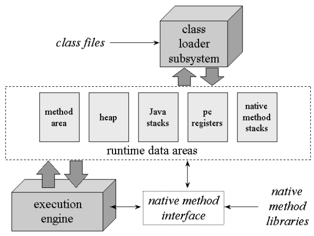
JVM Architecture



Program Counter

- Each thread of a running program has its own pc register, or program counter, which is created when the thread is started
  - The pc register is one word in size, so it can hold both a native pointer and a returnAddress
  - As a thread executes a Java method, the pc register contains the address of the current instruction being executed by the thread
  - An "address" can be a native pointer or an offset from the beginning of a method's bytecodes
  - a thread is executing a native method, the value of the pc register is undefined

JVM Architecture



Java Stack

- When a new thread is launched, the Java virtual machine creates a new Java stack for the thread
  - a Java stack stores a thread's state in discrete frames
    - only two operations directly on Java Stacks: push and pop frames
  - When a thread invokes a Java method, the virtual machine creates and pushes a new frame onto the thread's Java stack
    - This new frame then becomes the current frame
  - When a method completes, the Java virtual machine pops and discards the method's stack frame
    - The frame for the previous method then becomes the current frame
- All the data on a thread's Java stack is private to that thread

JVM Architecture

- Stack frame
  - The stack frame has three parts
    - local variables, operand stack, and frame data
  - When the Java virtual machine invokes a Java method, it creates a stack frame of the proper size for the method and pushes it onto the Java stack
- Local variables
  - The local variables section of the Java stack frame is organized as a zero-based array of words
    - Instructions that use a value from the local variables section provide an index into the zero-based array
    - Values of type int, float, reference, and returnAddress occupy one entry in the local variables array
    - Values of type byte, short, and char are converted to int before being stored into the local variables
    - Values of type long and double occupy two consecutive entries in the array

JVM Architecture

- Local variables
  - The local variables section contains a method's parameters and local variables

```
class ExampleLocalVar {
  public static int runClassMethod(int i,
    long l, float f, double d, Object o, byte b) {
    return 0;
  }
  public int runInstanceMethod(char c,
    double d, short s, boolean b) {
    return 0;
  }
}
```

runClassMethod()

index	type	parameter
0	int	int i
1	long	long l
3	float	float f
4	double	double d
6	reference	Object o
7	int	byte b

runInstanceMethod()

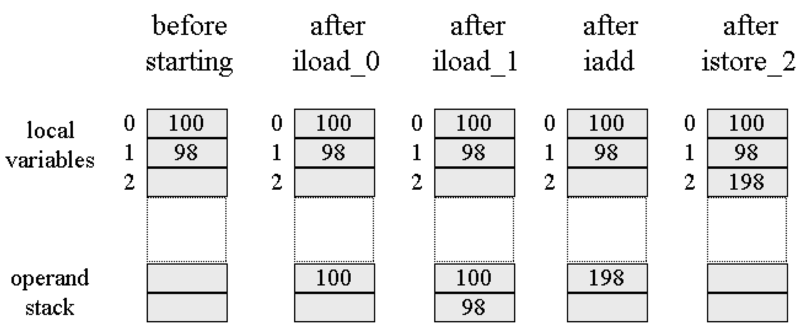
index	type	parameter
0	reference	hidden this
1	int	char c
2	double	double d
4	int	short s
5	int	boolean b

JVM Architecture

◆ Operand stack

- The Java virtual machine is stack-based
  - its instructions take their operands from the operand stack rather than from registers
  - The Java virtual machine uses the operand stack as a work space

```
iload_0    // push the int in local variable 0
iload_1    // push the int in local variable 1
iadd       // pop two ints, add them, push result
istore_2   // pop int, store into local variable 2
```

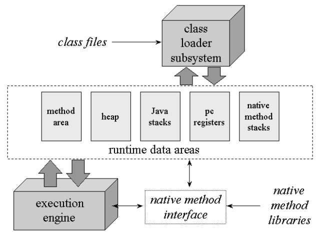


JVM Architecture

◆ Frame Data

- The Java stack frame includes data to support constant pool resolution, normal method return, and exception dispatch
  - stored in the *frame data* portion of the Java stack frame
- Many instructions in the Java virtual machine's instruction set refer to entries in the constant pool
  - Some instructions merely push constant values from the constant pool onto the operand stack
  - Some instructions use constant pool entries to refer to classes or arrays to instantiate, fields to access, or methods to invoke.
  - Other instructions determine whether a particular object is a descendant of a particular class or interface specified by a constant pool entry

JVM Architecture



◆ Execution Engine

- the behavior of the execution engine is defined in terms of an instruction set
  - For each instruction, the specification describes in detail *what* an implementation should do when it encounters the instruction as it executes bytecodes
    - Their implementations can interpret, just-in-time compile, execute natively in silicon, use a combination of these, or dream up some brand new technique
- Each thread of a running Java application is a distinct instance of the virtual machine's execution engine
  - A Java virtual machine implementation may use other threads invisible to the running application, such as a thread that performs garbage collection
    - Such threads need not be "instances" of the implementation's execution engine

JVM Architecture

◆ Instruction Set

- A method's bytecode stream is a sequence of instructions for the Java virtual machine
  - Each instruction consists of a one-byte *opcode* followed by zero or more *operands*
    - Many Java virtual machine instructions take no operands, and therefore consist only of an opcode
- An execution engine fetches an opcode and, if that opcode has operands, fetches the operands

```
class Act {
  public static void doMathForever() {
    int i = 0;
    for (;;) {
      i += 1;
      i *= 2;
    }
  }
}
```

```
// Bytecode: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Method void doMathForever()
0 iconst_0        // 03
1 istore_0        // 3b
2 iinc 0, 1       // 84 00 01
5 iload_0         // 1a
6 iconst_2        // 05
7 imul            // 68
8 istore_0        // 3b
9 goto 2          // a7 ff f9
```

JVM Architecture

- Instruction Set
  - The central focus of the Java virtual machine's instruction set is the operand stack
    - Values are generally pushed onto the operand stack before they are used
  - Several goals guided the design of the Java virtual machine's instruction set
    - platform independence
      - the stack-centered design--make it easier to implement the Java virtual machine on a wide variety of host architectures
    - network mobility
      - one major design consideration was class file compactness
    - security
      - the ability to do bytecode verification
      - The verification capability is needed as part of Java's security framework

Instruction Set Summary

- A Java virtual machine instruction consists of
  - a one-byte *opcode* specifying the operation to be performed
  - followed by zero or more *operands* supplying arguments or data that are used by the operation
  - Many instructions have no operands and consist only of an opcode
- The inner loop of a Java virtual machine interpreter is effectively

```
do {
        fetch an opcode;
        if (operands) fetch operands;
        execute the action for the opcode;
} while (there is more to do);
```

Types and JVM

- Most of the instruction in JVM instruction set encode type information
  - For the majority of typed instructions, the instruction type is represented explicitly in the opcode mnemonic by a letter
    - *i* for an int operation, *l* for long, *s* for short, *b* for byte, *c* for char, *f* for float, *d* for double, and *a* for reference
  - Some instructions for which the type is unambiguous do not have a type letter in their mnemonic
    - e.g. *arraylength* always operates on an object that is an array
  - Some instructions, such as *goto*, an unconditional control transfer, do not operate on typed operands
  - None have forms for the *boolean* type

Load and Store Instructions

- The load and store instructions transfer values between the local variables and the operand stack of a JVM frame
  - Load a local variable onto the operand stack
    - *iload, iload_⟨n⟩, lload, lload_⟨n⟩, fload, fload_⟨n⟩, dload, dload_⟨n⟩, aload, aload_⟨n⟩*
  - Store a value from the operand stack into a local variable
    - *istore, istore_⟨n⟩, lstore, lstore_⟨n⟩, fstore, fstore_⟨n⟩, dstore, dstore_⟨n⟩, astore, astore_⟨n⟩*
  - Load a constant onto the operand stack
    - *bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, iconst_m1, icont_⟨i⟩, lcont_⟨l⟩, fcont_⟨f⟩, dcont_⟨d⟩*
  - Gain access to more local variable using a wider index, or to a larger immediate operand
    - *wide*

Arithmetic Instructions

- The arithmetic instructions compute a result that is typically a function of two values on the operand stack, pushing the result back on the operand stack
  - There are two main kinds of arithmetic instructions
    - those operating on integer values, and
    - those operating on floating-point values
  - There is no direct support for integer arithmetic on values of the byte, short, and char types, or for values of the boolean type
    - those operations are handled by instructions operating on type int
  - Integer and floating-point instructions also differ in their behavior on overflow and divide-by-zero

Arithmetic Instructions

- The arithmetic instructions
  - Add
    - *iadd, ladd, fadd, dadd*
  - Subtract
    - *isub, lsub, fsub, dsub*.
  - Multiply
    - *imul, lmul, fmul, dmul*
  - Divide
    - *idiv, ldiv, fdiv, ddiv*
  - Remainder
    - *irem, lrem, frem, drem*
  - Negate
    - *ineg, lneg, fneg, dneg*
  - Local variable increment
    - *iinc*

Arithmetic Instructions

- The arithmetic instructions
  - Bitwise OR
    - *ior, lor*
  - Bitwise AND
    - *iand, land*
  - Bitwise exclusive OR
    - *ixor, lxor*
  - Comparison
    - *dcmpg, dcmpl, fcmpg, fcmpl, lcmp*
  - Shift
    - *ishl, ishr, iushr, lshl, lshr, lushr*

Type Conversion Instructions

- The type conversion instructions allow conversion between JVM numeric types.
  - These may be used to implement explicit conversions in user code or to mitigate the lack of orthogonality in the instruction set of the Java virtual machine
- The Java virtual machine directly supports the following widening numeric conversions:
  - int to long, float, or double
    - *i2l, i2f, i2d*
  - long to float or double
    - *l2f, l2d*
  - float to double
    - *f2d*

Type Conversion Instructions

- The JVM also directly supports the following narrowing numeric conversions:
  - int to byte, short, or char
    - *i2b, i2c, i2s*
  - long to int
    - *l2i*
  - float to int or long
    - *f2i, f2l*
  - double to int, long, or float
    - *d2i, d2l, d2f*

Object Creation and Manipulation

- Although both class instances and arrays are objects, the Java virtual machine creates and manipulates class instances and arrays using distinct sets of instructions
  - Create a new class instance
    - *new*.
  - Create a new array
    - *newarray, anewarray, multianewarray*
  - Access fields of classes (static fields, known as class variables) and fields of class instances (non-static fields, known as instance variables)
    - *getfield, putfield, getstatic, putstatic*

Object Creation and Manipulation

- Load an array component onto the operand stack
  - *baload, caload, saload, iaload, laload, faload, daload, aaload*.
- Store a value from the operand stack as an array component
  - *bastore, castore, sastore, iastore, lastore, fastore, dastore, aastore*.
- Get the length of array
  - *arraylength*.
- Check properties of class instances or arrays
  - *instanceof, checkcast*

Operand Stack Management Instructions

- Instructions are provided for the direct manipulation of the operand stack
  - Pop from stack
    - *pop, pop2*
  - Duplicate stack elements
    - *dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2*
  - Swap stack elements
    - *swap*

Control Transfer Instructions

- The control transfer instructions conditionally or unconditionally cause the JVM to continue execution with an instruction other than the one following the control transfer instruction
  - Conditional branch
    - *ifeq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpne*
  - Compound conditional branch
    - *tableswitch, lookupswitch*
  - Unconditional branch
    - *goto, goto_w, jsr, jsr_w, ret*

Method Invocation Instructions

- The following four instructions invoke methods:
  - *invokevirtual*

    invokes an instance method of an object, dispatching on the (virtual) type of the object. This is the normal method dispatch in the Java programming language.
  - *invokeinterface*

    invokes a method that is implemented by an interface, searching the methods implemented by the particular runtime object to find the appropriate method.
  - *invokespecial*

    invokes an instance method requiring special handling, whether an instance initialization method, a private method, or a superclass method.
  - *invokestatic*

    invokes a class (static) method in a named class

Method Return Instructions

- The method return instructions, which are distinguished by return type:
  - *ireturn*

    used to return values of type boolean, byte, char, short, and int
  - *lreturn*
  - *dreturn*
  - *areturn*
  - *return*

    returns from methods declared to be void

Synchronization

- The JVM supports synchronization of both methods and sequences of instructions within a method using a single synchronization construct
  - the *monitor*
- Method-level synchronization is handled as part of method invocation and return
- Synchronization of sequences of instructions is typically used to encode the synchronized blocks of the Java programming language
  - The JVM supplies the *monitorenter* and *monitorexit* instructions to support such constructs