The Java Virtual Machine
The Java Virtual Machine

“Java Architecture”

- Java Programming Language
- Java Virtual Machine (JVM)
- Java API
The Java Programming Environment

**Compile-time environment**
- Your program’s source files
  - A.java
  - B.java
  - C.java

  **Java compiler**
  - A.class
  - B.class
  - C.class

**Run-time environment**
- Your program’s class files
  - A.class
  - B.class
  - C.class

  **Java Virtual Machine**
  - Object.class
  - String.class
  - ...
Phases of Compilation

- Character stream
  - Scanner (lexical analysis)
- Token stream
  - Parser (syntax analysis)
- Parse tree
  - Semantic analysis
- Abstract syntax tree with annotations
  - Intermediate code generation
- Flow graph with pseudo-instructions in basic blocks
  - Machine-independent code improvement
- Modified flow graph
  - Target code generation
- (Almost) assembly language
  - Machine-specific code improvement
- Real assembly language
The Java Platform

The byte code generated by the Java front-end is an *intermediate form*

- Compact
- Platform-independent
The Class File

Java class file contains

• Byte code for data and methods (intermediate form, platform independent) (*remember byte code?)

• *Symbolic* references from one class file to another
  – Class names in text strings
  – Decompiling/reverse engineering quite easy

• Field names and descriptors (type info)

• Method names and descriptors (num args, arg types)

• Symbolic refs to other class methods/fields, own methods/fields
The Role of the Virtual Machine

The Java Virtual Machine

- Your program’s class files
  - Local or Remote
- class loader
- bytecodes
- execution engine
- native method invocations
- Host operating system
- The Java API’s class files
Class Loaders

- Bootstrap (default) loader (in the JVM)
- User-defined (custom) loaders
Dynamic Class Loading

• You don't have to know at compile-time all the classes that may ultimately take part in a running Java application.

User-defined class loaders enable you to dynamically extend a Java app at run-time

• As it runs, your app can determine what extra classes it needs and load them

• Custom loaders can download classes across a network (applets), get them out of some kind of database, or even calculate them on the fly.
The Execution Engine

Back-end transformation and execution

- **Simple JVM**
  - byte code interpretation

- **Just-in-time compiler**
  - Method byte codes are compiled into machine code the first time they are invoked
  - The machine code is cached for subsequent invocation
  - It requires more memory

- **Adaptive optimization**
  - The interpreter monitors the activity of the program, compiling the heavily used part of the program into machine code
  - It is much faster than simple interpretation, a little more memory
  - The memory requirement is only slightly larger due to the 20%/80% rule of program execution (In general, 20% of the code is responsible for 80% of the execution)
The Java Virtual Machine
Shared Data Areas

Each JVM has one of each:

✓ Method area: byte code and class (static) data storage
✓ Heap: object storage
Thread Data Areas

Frame in Execution
Data Types

Primitive Types
  - Numeric Types
    - boolean
    - returnAddress
  - Floating-Point Types
    - float
    - double
  - Integral Types
    - byte
    - short
    - int
    - long
    - char

Reference Types
  - reference
    - class types
    - interface types
    - array types
# Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>8-bit signed two's complement integer (-2⁷ to 2⁷ - 1, inclusive)</td>
</tr>
<tr>
<td>short</td>
<td>16-bit signed two's complement integer (-2¹⁵ to 2¹⁵ - 1, inclusive)</td>
</tr>
<tr>
<td>int</td>
<td>32-bit signed two's complement integer (-2³¹ to 2³¹ - 1, inclusive)</td>
</tr>
<tr>
<td>long</td>
<td>64-bit signed two's complement integer (-2⁶³ to 2⁶³ - 1, inclusive)</td>
</tr>
<tr>
<td>char</td>
<td>16-bit unsigned Unicode character (0 to 2¹⁶ - 1, inclusive)</td>
</tr>
<tr>
<td>float</td>
<td>32-bit IEEE 754 single-precision float</td>
</tr>
<tr>
<td>double</td>
<td>64-bit IEEE 754 double-precision float</td>
</tr>
<tr>
<td>returnAddress</td>
<td>address of an opcode within the same method</td>
</tr>
<tr>
<td>reference</td>
<td>reference to an object on the heap, or null</td>
</tr>
</tbody>
</table>
Class Loader Subsystem

• Loading, Linking and Initialization
  1. Loading: finding and importing the binary data for a type
  2. Linking: performing verification, preparation, and (optionally) resolution
     a) Verification: ensuring the correctness of the imported type
     b) Preparation: allocating memory for class variables and initializing the memory to default values
     c) Resolution: transforming symbolic references from the type into direct references.
  3. Initialization: invoking Java code that initializes class variables to their proper starting values.
Class Loader Subsystem

- The Bootstrap Class Loader
  - Java virtual machine implementations must be able to recognize and load classes and interfaces.
  - An implementation is free to recognize other binary forms besides class files, but it must recognize class files.
  - Every Java virtual machine implementation has a bootstrap class loader, which knows how to load trusted classes, including the classes of the Java API.
  - The Java virtual machine specification doesn't define how the bootstrap loader should locate classes. That is another decision the specification leaves to implementation designers.
Class Loader Subsystem

• User-Defined Class Loaders
  – Although user-defined class loaders themselves are part of the Java application, the methods below in class ClassLoader are gateways into JVM:
    
    protected final Class defineClass();
    protected final Class findSystemClass();
    protected final void resolveClass();

  – Any Java virtual machine implementation must take care to connect these methods of class ClassLoader to the internal class loader subsystem.
Class Loader Subsystem

• Name Spaces
  – Each class loader maintains its own name space populated by the types it has loaded.
  – Because each class loader has its own name space, a single Java application can load multiple types with the same fully qualified name.
  – A type's fully qualified name, therefore, is not always enough to uniquely identify it inside a Java virtual machine instance.
  – The identity of the class loader that loaded the type (the identity of the name space it is in) may also be needed to uniquely identify that type.
Stack Frames

Stack frames have three parts

• **Local variables**
  – The local variables section of the Java stack frame is organized as a zero-based array of words.

• **Operand stack**
  – Array-based implementation of simple stack.

• **Frame data**
  – Data to support constant pool resolution, normal method return, and exception dispatch.
```java
class Example3a {
    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;
    }
}
```

### Stack Frame

**Local Variables**

<table>
<thead>
<tr>
<th>index</th>
<th>type</th>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>int</td>
<td>int i</td>
</tr>
<tr>
<td>1</td>
<td>long</td>
<td>long l</td>
</tr>
<tr>
<td>3</td>
<td>float</td>
<td>float f</td>
</tr>
<tr>
<td>4</td>
<td>double</td>
<td>double d</td>
</tr>
<tr>
<td>6</td>
<td>reference</td>
<td>Object o</td>
</tr>
<tr>
<td>7</td>
<td>int</td>
<td>byte b</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>index</th>
<th>type</th>
<th>parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>reference</td>
<td>hidden this</td>
</tr>
<tr>
<td>1</td>
<td>int</td>
<td>char c</td>
</tr>
<tr>
<td>2</td>
<td>double</td>
<td>double d</td>
</tr>
<tr>
<td>4</td>
<td>int</td>
<td>short s</td>
</tr>
<tr>
<td>5</td>
<td>int</td>
<td>boolean b</td>
</tr>
</tbody>
</table>
Stack Frame
Operand Stack

Adding 2 numbers

```
iload_0
iload_1
Iadd
istore_2
```

Compiler can tell how many slots the op stack will need for a method

<table>
<thead>
<tr>
<th></th>
<th>before starting</th>
<th>after iload_0</th>
<th>after iload_1</th>
<th>after iadd</th>
<th>after istore_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>local variables</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>98</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

operand stack

```
100
100
98
198
```
The stack frame also supports

- Constant pool resolution
- Normal method return
- Exception dispatch
class Example3c {
    public static void addAndPrint() {
        double result =
            addTwoTypes(1, 88.88);
        System.out.println(result);
    }
    public static double addTwoTypes(int i, double d) {
        return i + d;
    }
}
A simulated stack of the target language (e.g. C) is created for JNI
The Heap

• Class instances (objects) and arrays are stored in a single, shared heap

• Each Java application has its own heap
  – Isolation
  – But a JVM crash will break this isolation

• JVM heaps always implement garbage collection mechanisms
Heap
Monolithic Object Representation

an object reference
ptr into heap

the heap

ptr to class data
instance data
instance data
instance data
instance data

the method area

class data
The Heap
Split Object Representation

The heap

- the handle pool
  - ptr into object pool
  - ptr to class data

- the object pool
  - instance data
  - instance data
  - instance data

the method area

class data
The Heap
Memory/Speed Tradeoff

- The heap
- An object reference
  - ptr into heap
- The method area
  - Method table
  - ptr to full class data
  - ptr to method data
  - ptr to method data
  - ptr to method data
- Entry point into all data for the class
- Method data
  - Method data
  - Method data
  - Method data

- ptr to special structure
  - instance data
  - instance data
The Heap
Arrays as Objects

```java
int[][] ar =
    new int[2][2];

ar (an array ref)
```

The heap

```
ptr to class data
length (= 2)
-ar[0][0] (an array ref)
-ar[0][1] (an array ref)
-ar[1][0] (an array ref)
-ar[1][1] (an array ref)
```

The method area

```
class data for [I
-class data for [I
```
Reference

The content of this lecture is based on *Inside the Java 2 Virtual Machine* by Bill Venner

- Chapter 1 Introduction to Java's Architecture

- Chapter 5 The Java Virtual Machine

- Interactive Illustrations
Examples

HeapOfFish

– http://www.artima.com/insidejvm/applets/HeapOfFish.html
– Object allocation illustration

Eternal Math Example

– JVM execution, operand stack, illustration