Java Implementation

- Compiler and Virtual Machine
  - Compiler produces bytecode
  - Virtual machine loads classes on demand, verifies bytecode properties, interprets bytecode
- Why this design?
  - Bytecode interpreter/compilers used before
    - Pascal "pcode": Smalltalk compilers use bytecode
  - Minimize machine-dependent part of implementation
    - Do optimization on bytecode when possible
    - Keep bytecode interpreter simple
  - For Java, this gives portability
    - Transmit bytecode across network

Java Virtual Machine Architecture

Class loader

- Runtime system loads classes as needed
  - When class is referenced, loader searches for file of compiled bytecode instructions
- Default loading mechanism can be replaced
  - Define alternate ClassLoader object
    - Extend the abstract ClassLoader class and implementation
    - ClassLoader does not implement abstract method loadClass, but has methods that can be used to implement loadClass
  - Can obtain bytecodes from alternate source
    - VM restricts applet communication to site that supplied applet
Example issue in class loading and linking:
Static members and initialization

```java
class ... {
    /* static variable with initial value */
    static int x = initial_value
    /* ---- static initialization block ---- */
    static {
        /* code executed once, when loaded */
    }
}
```

- Initialization is important
  - Cannot initialize class fields until loaded
- Static block cannot raise an exception
  - Handler may not be installed at class loading time

JVM Linker and Verifier

- **Linker**
  - Adds compiled class or interface to runtime system
  - Creates static fields and initializes them
  - Resolves names
    - Checks symbolic names and replaces with direct references
- **Verifier**
  - Check bytecode of a class or interface before loaded
  - Throw VerifyError exception if error occurs

Verifier

- Bytecode may not come from standard compiler
  - Evil hacker may write dangerous bytecode
- **Verifier** checks correctness of bytecode
  - Every instruction must have a valid operation code
  - Every branch instruction must branch to the start of some other instruction, not middle of instruction
  - Every method must have a structurally correct signature
  - Every instruction obeys the Java type discipline

  Last condition is fairly complicated.

Bytecode interpreter

- Standard virtual machine interprets instructions
  - Perform run-time checks such as array bounds
  - Possible to compile bytecode class file to native code
- Java programs can call native methods
  - Typically functions written in C
- Multiple bytecodes for method lookup
  - invokevirtual - when class of object known
  - invokeinterface - when interface of object known
  - invokestatic - static methods
  - invokespecial - some special cases

Type Safety of JVM

- Run-time type checking
  - All casts are checked to make sure type safe
  - All array references are checked to make sure the array index is within the array bounds
  - References are tested to make sure they are not null before they are dereferenced.

Additional features
- Automatic garbage collection
- No pointer arithmetic
  - If program accesses memory, that memory is allocated to the program and declared with correct type

JVM uses stack machine

- **Java**
  - Class A extends Object {
    - void f(int val) { i = val + 1; }
  }

- **Bytecode**
  - Method void f(int)
    - aload 0 ; object ref this
    - iload 1 ; int val
    - iconst 1
    - iadd ; add val +1
    - putfield #4 <Field int i>
    - return

  The above bytecode is a JVM activation record that refers to const pool.
Field and method access

- Instruction includes index into constant pool
  - Constant pool stores symbolic names
  - Store once, instead of each instruction, to save space
- First execution
  - Use symbolic name to find field or method
- Second execution
  - Use modified “quick” instruction to simplify search

invokeinterface <method-spec>

- Sample code
  ```java
  void add2(Incrementable x) { x.inc(); x.inc(); }
  ```
- Search for method
  - find class of the object operand (operand on stack)
  - find method with the given name and signature
- Call the method
  - Usual function call with new activation record, etc.

Why is search necessary?

```java
interface A {
   public void f();
}
interface B {
   public void g();
   public void g();
}
class C implements A, B {
   ...
}
```

Class C cannot have method f first and method g first

invokevirtual <method-spec>

- Similar to invokeinterface, but class is known
- Search for method
  - search the method table of this class
  - find method with the given name and signature
- Can we use static type for efficiency?
  - Each execution of an instruction will be to object from subclass of statically-known class
  - Constant offset into vtable
    - like C++, but dynamic linking makes search useful first time
  - See next slide

Bytecode rewriting: invokevirtual

- After search, rewrite bytecode to use fixed offset into the vtable. No search on second execution.

Bytecode rewriting: invokeinterface

- Cache address of method; check class on second use
Bytecode Verifier

- Let’s look at one example to see how this works
- Correctness condition
  - No operations should be invoked on an object until it has been initialized
- Bytecode instructions
  - new (class) allocate memory for object
  - init (class) initialize object on top of stack
  - use (class) use object on top of stack
  (idealization for purpose of presentation)

Object creation

- Example:
  ```java
  Point p = new Point(3)
  ```
  - Java source
  - bytecode

- No easy pattern to match
- Multiple refs to same uninitialized object
  - Need some form of alias analysis

Alias Analysis

- Other situations:
  1: new P
  2: new P or
  3: init P
  - Equivalence classes based on line where object was created.

Tracking initialize-before-use

- Alias analysis uses line numbers
  - Two pointers to "initialized object created at line 47" are assumed to point to same object
  - All accessible objects must be initialized before jump backwards (possible loop)
- Oversight in treatment of local subroutines
  - Used in implementation of try-finally
  - Object created in finally not necessarily initialized
- No clear security consequence
  - Bug fixed
  - Have proved correctness of modified verifier for init

Aside: bytecodes for try-finally

- Idea
  - Finally clause implemented as lightweight subroutine
- Example code
  ```java
  static int f(boolean bVal) {
    try {
      if (bVal) { return 1; }
      return 0;
    } finally {
      System.out.println("About to return");
    }
  }
  ```
- Bytecode on next slide
  - Print before returning, regardless of which return is executed

Bytecode


```java
0 iload_0 // Push local variable 0 (arg passed as divisor)
1 ifeq 11 // Push local variable 1 (arg passed as dividend)
4 iconst_1 // Push int 1
5 istore_3 // Pop an int (the 1), store into local variable 3
6 jsr 24 // Jump to the mini-subroutine for the finally clause
9 iload 3 // Push local variable 3 (the 1)
10 ireturn // Return int on top of the stack (the 1)
12 iload_1 // Push local variable 1 (arg passed as divisor)
13 iload_0 // Push local variable 0 (arg passed as dividend)
22 astore_2 // Pop the return address, store it in local variable 2
25 getstatic #8 // Get a reference to java.lang.System.out
28 ldc #12 // Push <String "Got old fashioned."> from the constant pool
30 invokevirtual #7 // Invoke System.out.println()
33 ret 2 // Return to return address stored in local variable 2
```
Bug in Sun’s JDK 1.1.4

Example:

1: jsr 10
2: store 1
3: jsr 10
4: store 2
5: load 2
6: init P
7: load 1
8: use P
9: halt

Variables 1 and 2 contain references to two different objects which are both "uninitialized object created on line 11"

Bytecode verifier not designed for code that creates uninitialized object in jsr subroutine

Implementing Generics

Two possible implementations
- Heterogeneous: instantiate generics
- Homogeneous: translate generic class to standard class

Example for next few slides: generic list class

```cpp
template <type t> class List {
    private: t* data; List<t>* next;
    public: void Cons (t* x) { ... }  // Add a new element to the list
    t* Head ( ) { ... }  // Return the head of the list
    List<t> Tail ( ) { ... }  // Return the tail of the list
};
```

Issues

- Data on heap, manipulated by pointer (Java)
  - Every list cell has two pointers, data and next
  - All pointers are same size
  - Can use same representation, code for all types
- Data stored in local variables (C++)
  - List cell must have space for data
  - Different representation for different types
  - Different code if offset of fields built into code
- When is template instantiated?
  - Compile- or link-time (C++)
  - Java alternative: class load time - next few slides
  - Java Generics: no "Instantiation", but erasure at compile time
  - C#: just-in-time instantiation, with some code-sharing tricks ...

"Homogeneous Implementation"

Same representation and code for all types of data

"Heterogeneous Implementation"

Specialize representation, code according to type

Heterogeneous Implementation for Java

Compile generic class C<param>
- Check use of parameter type according to constraints
- Produce extended form of bytecode class file
  - Store constraints, type parameter names in bytecode file
- Expand when class C<actual> is loaded
  - Replace parameter type by actual class
  - Result is ordinary class file
  - This is a preprocessor to the class loader:
    - No change to the virtual machine
    - No need for additional bytecodes
Example: Hash Table

```java
interface Hashable {
   int HashCode();
};

class HashTable < Key implements Hashable, Value> {
   void Insert (Key k, Value v) {
      int bucket = k.HashCode();
      InsertAt (bucket, k, v);
   }
   ...
};
```

Generic bytecode with placeholders

```java
void Insert (Key k, Value v) {
   int bucket = k.HashCode();
   InsertAt (bucket, k, v);
}
```

Instantiation of generic bytecode

```java
void Insert (Key k, Value v) {
   int bucket = k.HashCode();
   InsertAt (bucket, k, v);
}
```

Loading parameterized class file

- Use of HashTable <Name, Integer> invokes loader
- Several preprocess steps
  - Locate bytecode for parameterized class, actual types
  - Check the parameter constraints against actual class
  - Substitute actual type name for parameter type
  - Proceed with verifier, linker as usual
- Can be implemented with \(-500\) lines Java code
  - Portable, efficient, no need to change virtual machine

Java 1.5 Implementation

- Homogeneous implementation
  ```java
class Stack {
   void push(Object o) { ... }
   Object pop() { ... }
}...
```  
- Algorithm
  - replace class parameter \(<A>\) by Object, insert casts
  - if \(<A>\) extends \(B\), replace \(A\) by \(B\)
- Why choose this implementation?
  - Backward compatibility of distributed bytecode
  - Surprise: sometimes faster because class loading slow

Some details that matter

- Allocation of static variables
  - Heterogeneous: separate copy for each instance
  - Homogenous: one copy shared by all instances
- Constructor of actual class parameter
  - Heterogeneous: class \(G<T>\) \(\rightarrow \) \(T x = new T\);  
    - Homogenous: new \(T\) may just be Object!
    - Creation of new object is not allowed in Java
- Resolve overloading
  - Heterogeneous: resolve at instantiation time (C++)
  - Homogenous: no information about type parameter
Example

- This Code is not legal java
  - class C<A> { A id (A x) {...} }
  - class D extends C<String> {
      Object id(Object x) {...}
  }
- Why?
  - Subclass method looks like a different method, but after erasure the signatures are the same

Outline

- Objects in Java
  - Classes, encapsulation, inheritance
- Type system
  - Primitive types, interfaces, arrays, exceptions
- Generics (added in Java 1.5)
  - Basics, wildcards, ...
- Virtual machine
  - Loader, verifier, linker, interpreter
  - Bytecodes for method lookup
  - Bytecode verifier (example: initialize before use)
- Implementation of generics
- Security issues

Java Security

- Security
  - Prevent unauthorized use of computational resources
- Java security
  - Java code can read input from careless user or malicious attacker
  - Java code can be transmitted over network – code may be written by careless friend or malicious attacker

Java is designed to reduce many security risks

Java Security Mechanisms

- Sandboxing
  - Run program in restricted environment
    - Analogy: child's sandbox with only safe toys
  - This term refers to
    - Features of loader, verifier, interpreter that restrict program
    - Java Security Manager, a special object that acts as access control "gatekeeper"
- Code signing
  - Use cryptography to establish origin of class file
    - This info can be used by security manager

Buffer Overflow Attack

- Most prevalent general security problem today
  - Large number of CERT advisories are related to buffer overflow vulnerabilities in OS, other code
- General network-based attack
  - Attacker sends carefully designed network msgs
  - Input causes privileged program (e.g., Sendmail) to do something it was not designed to do
- Does not work in Java
  - Illustrates what Java was designed to prevent

Sample C code to illustrate attack

```c
void f (char *str) {
    char buffer[16];
    ... 
    strcpy(buffer,str);
}

void main() {
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
      large_string[i] = 'A';
    f(large_string);
}
```

- Function
  - Copies str into buffer until null character found
  - Could write past end of buffer, over function return addr
- Calling program
  - Writes 'A' over f activation record
  - Function f "returns" to location 0x41414141
- This causes segmentation fault
- Variations
  - Put meaningful address in string
  - Put code in string and jump to it!!

See: Smashing the stack for fun and profit
Java Sandbox

- Four complementary mechanisms
  - **Class loader**
    - Separate namespaces for separate class loaders
    - Associates protection domain with each class
  - **Verifier and JVM run-time tests**
    - NO unchecked casts or other type errors, NO array overflow
    - Preserves private, protected visibility levels
  - **Security Manager**
    - Called by library functions to decide if request is allowed
    - Uses protection domain associated with code, user policy
    - Coming up in a few slides: stack inspection

Security Manager

- Java library functions call security manager
- Security manager object answers at run time
  - Decide if calling code is allowed to do operation
  - Examine protection domain of calling class
    - Signer: organization that signed code before loading
    - Location: URL where the Java classes came from
  - Uses the system policy to decide access permission

Sample SecurityManager methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>checkExec</td>
<td>Checks if the system commands can be executed.</td>
</tr>
<tr>
<td>checkRead</td>
<td>Checks if a file can be read from.</td>
</tr>
<tr>
<td>checkWrite</td>
<td>Checks if a file can be written to.</td>
</tr>
<tr>
<td>checkListen</td>
<td>Checks if a certain network port can be listened to for connections.</td>
</tr>
<tr>
<td>checkConnect</td>
<td>Checks if a network connection can be created.</td>
</tr>
<tr>
<td>checkCreate</td>
<td>Check to prevent the installation of additional ClassLoaders.</td>
</tr>
</tbody>
</table>

Stack Inspection

- Permission depends on
  - Permission of calling method
  - Permission of all methods above it on stack
    - Up to method that is trusted and asserts this trust

Stories: Netscape font / passwd bug; Shockwave plug-in

Java Summary

- **Objects**
  - have fields and methods
  - alloc on heap, access by pointer, garbage collected
- **Classes**
  - Public, Private, Protected, Package (not exactly C++)
  - Can have static (class) members
  - Constructors and finalize methods
- **Inheritance**
  - Single inheritance
  - Final classes and methods

Java Summary (II)

- **Subtyping**
  - Determined from inheritance hierarchy
- **Virtual machine**
  - Load bytecode for classes at run time
  - Verifier checks bytecode
  - Interpreter also makes run-time checks
    - type casts
    - array bounds
  - Portability and security are main considerations
Some Highlights

- **Dynamic lookup**
  - Different bytecodes for by-class, by-interface
  - Search vtable + Bytecode rewriting or caching
- **Subtyping**
  - Interfaces instead of multiple inheritance
  - Awkward treatment of array subtyping (my opinion)
- **Generics**
  - Type checked, not instantiated, some limitations (<T>…new T)
- **Bytecode-based JVM**
  - Bytecode verifier
  - Security: security manager, stack inspection

Comparison with C++

- **Almost everything is object** + Simplicity - Efficiency
  - except for values from primitive types
- **Type safe** + Safety +/- Code complexity - Efficiency
  - Arrays are bounds checked
  - No pointer arithmetic, no unchecked type casts
  - Garbage collected
- **Interpreted** + Portability + Safety - Efficiency
  - Compiled to byte code: a generalized form of assembly language designed to interpret quickly.
  - Byte codes contain type information

Comparison (cont’d)

- **Objects accessed by ptr** + Simplicity - Efficiency
  - No problems with direct manipulation of objects
- **Garbage collection**: + Safety + Simplicity - Efficiency
  - Needed to support type safety
- **Built-in concurrency support** + Portability
  - Used for concurrent garbage collection (avoid waiting?)
  - Concurrency control via synchronous methods
  - Part of network support: download data while executing
- **Exceptions**
  - As in C++, integral part of language design