CS 242

Java

John Mitchell

Outline

- Language Overview
  - History and design goals
- Classes and Inheritance
  - Object features
  - Encapsulation
  - Inheritance
- Types and Subtyping
  - Primitive and ref types
  - Interfaces; arrays
  - Exception hierarchy
  - Subtype polymorphism and generic programming
- Java virtual machine
  - Loader and initialization
  - Linker and verifier
  - Bytecode interpreter
- Method lookup
  - four different bytecodes
- Security
  - Buffer overflow
  - Java "sandbox"
  - Type safety and attacks

History

- James Gosling and others at Sun, 1990 - 95
- Oak language for "set-top box"
  - small networked device with television display
    - graphics
    - execution of simple programs
    - communication between local program and remote site
    - no "expert programmer" to deal with crash, etc.
- Internet application
  - simple language for writing programs that can be transmitted over network

Gates Saw Java as Real Threat

Publicly, Microsoft chief Bill Gates was nearly dismissive when he talked in 1996 about Sun Microsystems' Java programming language. But in internal company discussions, he wrote to staff members that Java and the threat the cross-platform technology posed to his company's Windows operating systems "scares the hell out of me."

Wired News Report
8:09 a.m. 22.Oct.98.PDT
(material from '98 trial)

Design Goals

- Portability
  - Internet-wide distribution: PC, Unix, Mac
- Reliability
  - Avoid program crashes and error messages
- Safety
  - Programmer may be malicious
- Simplicity and familiarity
  - Appeal to average programmer; less complex than C++
- Efficiency
  - Important but secondary

General design decisions

- Simplicity
  - Almost everything is an object
  - All objects on heap, accessed through pointers
  - No functions, no multiple inheritance, no go to, no operator overloading, no automatic coercions
- Portability and network transfer
  - Bytecode interpreter on many platforms
- Reliability and Safety
  - Typed source and bytecode language
  - Run-time type and bounds checks
  - Garbage collection

Java System

- The Java programming language
- Compiler and run-time system
  - Programmer compiles code
  - Compiled code transmitted on network
  - Receiver executes on interpreter (JVM)
  - Safety checks made before/during execution
- Library, including graphics, security, etc.
  - Large library made it easier for projects to adopt Java
  - Interoperability
    - Provision for “native” methods

Java Classes and Objects

- Syntax similar to C++
- Object
  - has fields and methods
  - is allocated on heap, not run-time stack
  - accessible through reference (only ptr assignment)
  - garbage collected
- Dynamic lookup
  - Similar in behavior to other languages
  - Static typing => more efficient than Smalltalk
  - Dynamic linking, interfaces => slower than C++

Sample Program

```
public class HelloWorld {
   public static void main(String[ ] args) {
      System.out.println{“Hello World!”);
   }
}
```

Static method = class method

Function can be called without creating object of the class

Point Class

```
class Point {
    private int x;
    protected void setX (int y)  {x = y;}
    public int  getX()      {return x;}
    Point(int xval) {x = xval;}       // constructor
};
```

- Visibility similar to C++, but not exactly (next slide)

Language Terminology

- Class, object - as in other languages
- Field – data member
- Method - member function
- Static members - class fields and methods
- this - self
- Package - set of classes in shared namespace
- Native method - method written in another language, typically C

Object initialization

- Java guarantees constructor call for each object
  - Memory allocated
  - Constructor called to initialize memory
  - Some interesting issues related to inheritance
    - We'll discuss later ...
- Cannot do this (would be bad C++ style):
  - Obj* obj = (Obj*)malloc(sizeof(Obj));

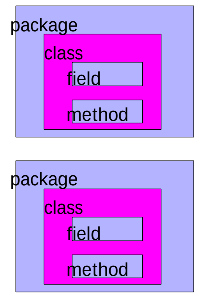
    use new instead ...
- Static fields of class initialized at class load time
  - Talk about class loading later

Garbage Collection and Finalize

- Objects are garbage collected
  - No explicit *free*
  - Avoid dangling pointers, resulting type errors
- Problem
  - What if object has opened file or holds lock?
- Solution
  - *finalize* method, called by the garbage collector
    - Before space is reclaimed, or when virtual machine exits
    - Space overflow is not really the right condition to trigger finalization when an object holds a lock...)
  - Important convention: call super.finalize

Encapsulation and packages

- Every field, method belongs to a class
- Every class is part of some package
  - Can be unnamed default package
  - File declares which package code belongs to



Visibility and access

- Four visibility distinctions
  - public, private, protected, package
- Method can refer to
  - private members of class it belongs to
  - non-private members of all classes in same package
  - protected members of superclasses (in diff package)
  - public members of classes in visible packages
    Visibility determined by files system, etc. (outside language)
- Qualified names (or use import)
  - java.lang.String.substring()

    package class method

Inheritance

- Similar to Smalltalk, C++
- Subclass inherits from superclass
  - Single inheritance only (but see interfaces)
- Some additional features
  - Conventions regarding *super* in constructor and *finalize* methods
  - Final classes and methods

Example subclass

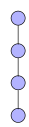
```
class ColorPoint extends Point {
   // Additional fields and methods
    private Color c;
    protected void setC (Color d)  {c = d;}
    public Color  getC()     {return c;}
   // Define constructor
    ColorPoint(int xval, Color cval) {
        super(xval);      // call Point constructor
        c = cval;  }      // initialize ColorPoint field
 };
```

Constructors and Super

- Java guarantees constructor call for each object
- This must be preserved by inheritance
  - Subclass constructor must call super constructor
    - If first statement is not call to super, then call super() inserted automatically by compiler
    - If superclass does not have a constructor with no args, then this causes compiler error (yuck)
    - Exception to rule: if one constructor invokes another, then it is responsibility of second constructor to call super, e.g.,
      ColorPoint() { ColorPoint(0,blue);}
      is compiled without inserting call to super
- Different conventions for finalize and super
  - Compiler does not force call to super finalize

Final classes and methods

- Restrict inheritance
  - Final classes and methods cannot be redefined
- Example
  - java.lang.System
- Reasons for this feature
  - Important for security
    - Programmer controls behavior of all subclasses
    - Critical because subclasses produce subtypes
  - Compare to C++ virtual/non-virtual
    - Method is "virtual" until it becomes final



Class *Object*

- Every class extends another class
  - Superclass is *Object* if no other class named
- Methods of class *Object*
  - GetClass – return the Class object representing class of the object
  - ToString – returns string representation of object
  - equals – default object equality (not ptr equality)
  - hashCode
  - Clone – makes a duplicate of an object
  - wait, notify, notifyAll – used with concurrency
  - finalize

Java Types and Subtyping

- Primitive types – *not* objects
  - Integers, Booleans, etc
- Reference types
  - Classes, interfaces, arrays
  - No syntax distinguishing Object * from Object
- Type conversion
  - If A <: B, and B x, then can cast x to A
  - Casts checked at run-time, may raise exception

Class and Interface Subtyping

- Class subtyping similar to C++
  - Statically typed language
  - Subclass produces subtype
  - Single inheritance => subclasses form tree
- Interfaces
  - Completely abstract classes
    - no implementation
    - Java also has abstract classes (without full impl)
  - Multiple subtyping
    - Interface can have multiple subtypes

Example

```
interface Shape {
   public float center();
   public void rotate(float degrees);
}
interface Drawable {
   public void setColor(Color c);
   public void draw();
}
class Circle implements Shape, Drawable {
   // does not inherit any implementation
   // but must define Shape, Drawable methods
}
```

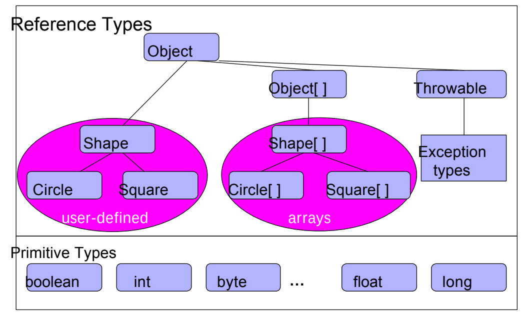
Properties of interfaces

- Flexibility
  - Allows subtype graph instead of tree
  - Avoids problems with multiple inheritance of implementations (remember C++ "diamond")
- Cost
  - Offset in method lookup table not known at compile
  - Different bytecodes for method lookup
    - one when class is known
    - one when only interface is known
      - search for location of method
      - cache for use next time this call is made (from this line)

Array types

- Automatically defined
  - Array type T[ ] exists for each class, interface type T
  - Cannot extended array types (array types are final)
  - Multi-dimensional arrays as arrays of arrays: T[ ] [ ]
- Reference type
  - An array variable is a pointer to an array, can be null
  - Example: Circle[] x = new Circle[array_size]
  - Anonymous array expression: new int[] {1,2,3, ... 10}
- Every array type is a subtype of Object[ ], Object
  - Length of array is not part of its static type

Classification of Java types



Array subtyping

- Covariance
  - if S <: T then S[ ] <: T[ ]
- Standard type error

```
class A {...}
class B extends A {...}
B[ ] bArray = new B[10]
A[ ] aArray = bArray     // considered OK since B[] <: A[]
aArray[0] = new A()      // allowed but run-time type error
                         // raises ArrayStoreException
```

Covariance problem again ...

- Remember Simula problem
  - If A <: B, then A ref <: B ref
  - Needed run-time test to prevent bad assignment
  - Covariance for assignable cells is not right in principle
- Explanation
  - interface of “T reference cell” is
    - put : $T \rightarrow T$ ref
    - get : T ref $\rightarrow T$
  - Remember covariance/contravariance of functions

Afterthought on Java arrays

Date: Fri, 09 Oct 1998 09:41:05 -0600
From: bill joy
Subject: ...[discussion about java genericity]

actually, java array covariance was done for less noble reasons ...: it made some generic "bcopy" (memory copy) and like operations much easier to write...
I proposed to take this out in 95, but it was too late (...).
i think it is unfortunate that it wasn't taken out...
it would have made adding genericity later much cleaner, and [array covariance] doesn't pay for its complexity today.
wnj

But compare this to C++!!

- Access by pointer: you can't do array subtyping.

```
B* barr[15];
A* aarr[] = barr;   // not allowed
```

- Direct naming: allowed, but you get garbage !!

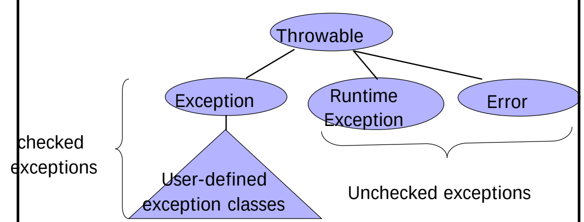
```
B barr[15];
A aarr[] = barr;
```

aarr[k] translates to *(aarr+sizeof(A)*k)
barr[k] translates to *(barr+sizeof(B)*k)
If sizeof(B) != sizeof(A), you just grab random bits.
Is there any sense to this?

Java Exceptions

- Similar basic functionality to ML, C++
  - Constructs to *throw* and *catch* exceptions
  - Dynamic scoping of handler
- Some differences
  - An exception is an object from an exception class
  - Subtyping between exception classes
    - Use subtyping to match type of exception or pass it on …
    - Similar functionality to ML pattern matching in handler
  - Type of method includes exceptions it can throw
    - Actually, only subclasses of Exception (see next slide)

Exception Classes



- If a method may throw a checked exception, then this must be in the type of the method

Try/finally blocks

- Exceptions are caught in try blocks

```
try {
    statements
}   catch (ex-type1 identifier1) {
            statements
} catch (ex-type2 identifier2) {
            statements
}  finally {
            statements
}
```

- Implementation: finally compiled to jsr

Why define new exception types?

- Exception may contain data
  - Class Throwable includes a string field so that cause of exception can be described
  - Pass other data by declaring additional fields or methods
- Subtype hierarchy used to catch exceptions

  catch <exception-type> <identifier> { … }

  will catch any exception from any subtype of exception-type and bind object to identifier

Java Generic Programming

- Java has class Object
  - Supertype of all object types
  - This allows “subtype polymorphism”
    - Can apply operation on class T to any subclass S <: T
- Java does not have templates
  - No parametric polymorphism
  - Many consider this the biggest deficiency of Java
- Java type system does not let you cheat
  - Can cast from supertype to subtype
  - Cast is checked at run time

Example generic construct: Lists

- Lists possible for any type of object
  - For any type t, can have type list_of_t
  - Operations cons, head, tail work for any type
- Define C++ generic list class

```
template <type t> class List {
        private: t* data; List<t> * next;
        public: void      Cons (t* x) { … }
                 t*       Head (    ) { … }
                 List<t>  Tail  (    ) { … }
};
```

Example generic construct: Stack

- Stack possible for any type of object
  - For any type t, can have type stack_of_t
  - Operations push, pop work for any type
- Define C++ generic list class

```
template <type t> class Stack {
        private: t data;  Stack<t> * next;
        public: void        push (t* x) { … }
                    Stack<t>* pop   (    ) { … }
};
```

- No equivalent Java mechanism

Current Java vs Templates

class Stack {	class Stack<A> {
void push(Object o) { ... }	void push(A a) { ... }
Object pop() { ... }	A pop() { ... }
...}	...}
String s = "Hello";	String s = "Hello";
Stack st = new Stack();	Stack<String> st = new Stack<String>();
…	
st.push(s);	st.push(s);
…	…
s = (String) st.pop();	s = st.pop();

Why no templates in Java?

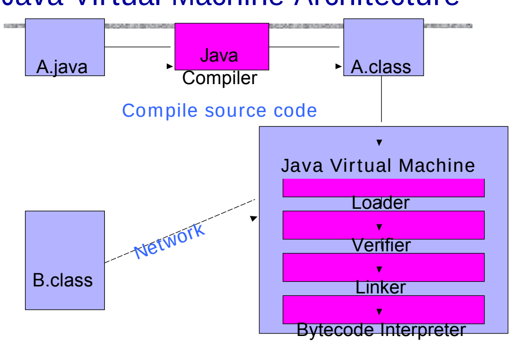
- Many proposals
- Basic language goals seem clear
- Details need to be worked out
  - Exact typing constraints
  - Implementation
    - Existing virtual machine?
    - Additional bytecodes?
    - Duplicate code for each instance?
    - Use same code (with casts) for all instances?

There is a Java Community proposal to add generics

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

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 for class or interface before loaded
  - Throw VerifyError exception if error occurs

Example issue in class loading and linking:

Static members and initialization

```
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

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

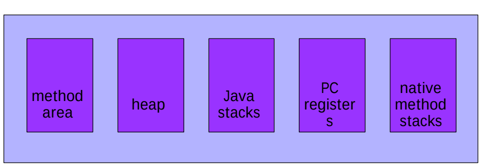
Last condition is fairly complicated

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

JVM memory areas

- Java program has one or more threads
- Each thread has its own stack
- All threads share same heap



JVM uses stack machine

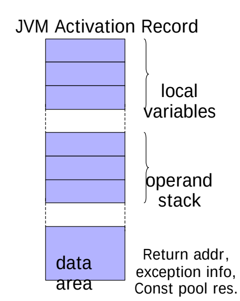
- Java

```
Class A extends Object {
    int i
    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
```

refers to const pool



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, the memory is allocated to the program and declared with correct type

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

  void add2(Incrementable x) { x.inc(); x.inc(); }
- Search for method
  - find class of the object operand (operand on stack)
    - must implement the interface named in <method-spec>
  - search the method table for this class
  - find method with the given name and signature
- Call the method
  - Usual function call with new activation record, etc.

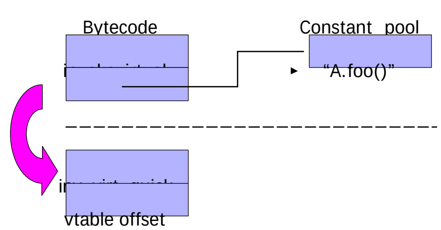
Why is search necessary?

```
interface Incrementable {
   public void inc();
}
class IntCounter implements Incrementable {
   public void add(int);
   public void inc();
   public int value();
}
class FloatCounter implements Incrementable {
   public void inc();
   public void add(float);
   public float value();
}
```

invokevirtual <method-spec>

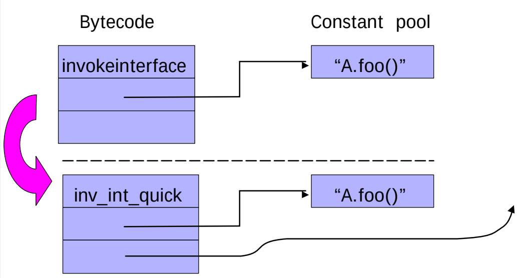
- Similar to invokeinterface, but class is known
- Search for method
  - search the method table for this class
  - find method with the given name and signature
- Can we use static type for efficiency?
  - Each execution of an instruction will be to object from subclass of statically-known class
  -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

Bytecode rewriting: invokevirtual



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

Bytecode rewriting: invokeinterface



Cache address of method; check class on second use

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

General Security Risks

- Denial of Service
  - Tie up your CPU, network connection, subnet, …
- Steal private information
  - User name, email address, password, credit card, …
- Compromise your system
  - Erase files, introduce virus, …

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 determine who wrote (or shipped) class file
    - This info can be used by security manager

Buffer Overflow Attack

- Most prevalent security problem today
  - Approximately 80% of CERT advisories are related to buffer overflow vulnerabilities in OS, other code
- Generally 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
  - This example illustrates what Java was designed to prevent

Sample code to illustrate idea

```
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 retun addr*
- Calling program
  - Writes 'A' over f activation record
  - Function f "returns" to location 0x4141414141
  - 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
    - Recall: stack inspection problem on midterm

Why is typing a security feature?

- Sandbox mechanisms all rely on type safety
- Example
  - Unchecked cast lets applet make any system call

```
int (*fp)()    /* variable "fp" is a function pointer       */
...
fp = addr;     /* assign address stored in an integer var   */
(*fp)(n);      /* call the function at this address         */
```

Other examples using Java type confusion in reader

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
  - Class may implement multiple interfaces
- 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

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