From arrangements: non-objective objects
http://www.richardbeenen.com/id3.html

Implementing Objects

Week 10
CS 212 – Spring 2008

Announcements

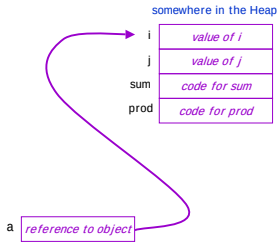
- Project Part 3
  - Part 3 code is due Thursday, April 10
- If you wish, you may schedule an appointment to talk to the TAs about your Design Document

Intuitive View of an Object

```
class A {
  int i, j;

  A (int ii, int jj) {
        i = ii; j = jj;
        }
  int sum () {
        return i + j;
        }
  int prod () {
        return i * j;
        }
  }

a = new A(4, 8);
```



This is close to what's actually done except we don't really store the code with the object

Variables within Classes

- Local variables (i.e., local to a method) reside on the stack, just as for functions
  - Location is *FBR+offset*
- Instance variables (i.e., fields) are stored within the object
  - Location is *objectAddress+offset*

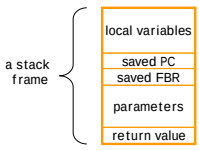
Calling a Method

- Basically, a method is just a function
  - Build a standard stack frame
  - Include one extra parameter: the object
- In other words, if the code is

  a.sum()

  then the extra parameter is a

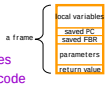
  (Actually, the address of a)



Function Call vs. Method Call

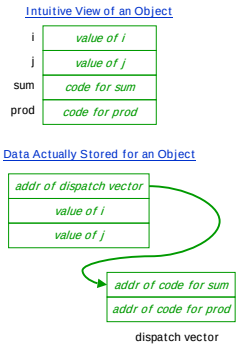
- Caller:
  - Push space for ret value
  - Push arguments
  - Push/update FBR
  - Push/update PC
- Callee:
  - Push local variables
  - Execute callee code
  - Clear local variables
  - Pop/restore PC
- Caller:
  - Pop/restore FBR
  - Clear arguments
  - (Ret value is left on stack)

- Caller:
  - Push space for ret value
  - Push object's address
  - Push arguments
  - Push/update FBR
  - Push/update PC
- Callee:
  - Push local variables
  - Execute method code
  - Clear local variables
  - Pop/restore PC
- Caller:
  - Pop/restore FBR
  - Clear arguments
  - Clear object's address
  - (Ret value is left on stack)



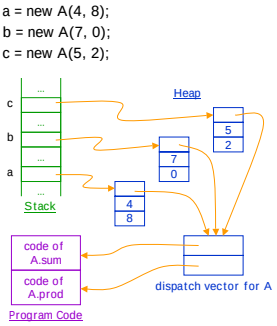
Dispatch Vector

- For a method, we don't store a copy of the method's code with each class instance
  - Instead we can store the address of the method's code
- But each instance of a class refers to exactly the same set of methods
  - It's wasteful for each object to store an address for each of its methods
- Instead, we use a *dispatch vector*
  - A simple table of method addresses stored somewhere else in the Heap



Shared Data for a Class

- Instances of the same class share the same *dispatch vector*
- This implies that a *dispatch vector* must be created for *each class*
- If there are static variables (i.e., class variables)
  - These would be stored in a *Static Data Area* with the dispatch vector
  - There would be one such *Static Data Area* for each class



What Info is Needed to Generate Code?

- For a local variable
  - Offset from FBR
- For a field
  - Address of object
  - Offset of field from start of object
- For a method
  - Address of object
    - From this, you can derive address of dispatch vector
  - Offset of method from start of dispatch vector
- All of this offset information is stored in the Symbol Table(s) (along with other information)
- For a field or a method
  - Address of object comes from local variable
    - Examples: a.i or a.sum()
  - Or address of object comes from hidden "this" parameter of a method
    - Examples: i or sum() when used within a method of A

Multiple Symbol Tables

- Program Symbol Table
  - Global variables
    - Type and location
  - Classes
    - Where to find class's dispatch vector
    - Size of corresponding object
  - Functions & constructors
    - Where to find corresponding code
    - Parameter types and return type
  - Need to know the above function and constructor information *before* generating any function or constructor code
    - Can use separate pass over the AST
- Class Symbol Table
  - Fields
    - Type & offset within object
  - Methods
    - Param types & return type
    - Offset within dispatch vector
  - *Private* fields and methods can be removed from table after class has been compiled
- Method/Function Symbol Table
  - Local variables
    - Type and offset from FBR
  - Entire table can be deleted after compiling the method or function

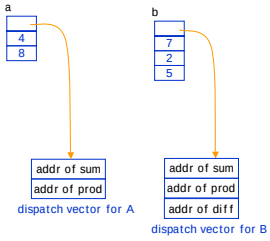
Inheritance

- We aren't doing inheritance for Bali Part 4, but here's how it works
- An object inherits all *public* fields and methods of its superclass
  - But the *private* fields and methods still exist
- When we create the code for a method, we don't know if we are using
  - An instance of the class itself
  - Or an instance of some subclass
- This implies that a subclass had better use the same offsets as its superclass
  - Same dispatch vector (with any new stuff at the end)
  - Same object layout (with any new stuff at the end)
- This allows a method's code to still work even though it's dealing with a subclass
  - Any "new stuff at the end" is never accessed by the method

Inheritance Example

```
class A {
   int i, j;
   A (int ii, int jj) {
        i = ii; j = jj;
        }
   int sum () {
        return i + j;
        }
   int prod () {
        return i * j;
        }
   }
class B extends A {
   int k;
   B (int ii, int jj) {
        super(ii, jj);
        k = i - j;
        }
   int diff () {
        return k;
        }
   }
```

```
a = new A(4, 8);
b = new B(7, 2);
x = b.prod();          // Uses A's code
```

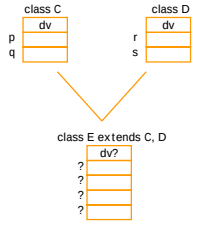


Overriding vs. Shadowing

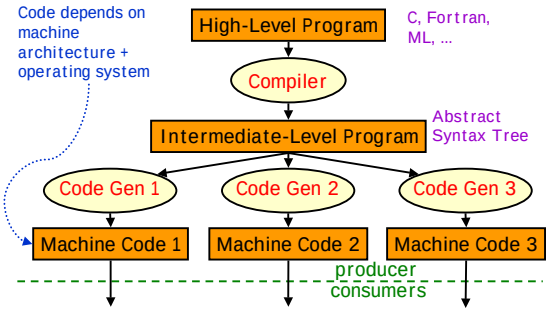
- In Java, what happens if a subclass redefines fields or methods that exist in the superclass?
- A method with the same signature will *override* the superclass's method
  - In other words, an instance of the subclass should call the *new method*, not the old one
  - This can be done by altering the dispatch vector
    - In the subclass's dispatch vector, the address of the new code *replaces* the address of the old code
- A field with the same name will *shadow* the superclass's field
  - In other words, the variable accessed depends on where the code is
  - This can be done by appending the new field on the end of the object layout (just as if the name were completely new)
    - The Symbol Table for the subclass knows only about the new field

Multiple Inheritance

- Java allows a class to inherit from at most one other class
- Other languages allow multiple inheritance
  - It becomes difficult to make offsets match for both the object layout and the dispatch vector
  - Other schemes are used



Compiling for Different Platforms



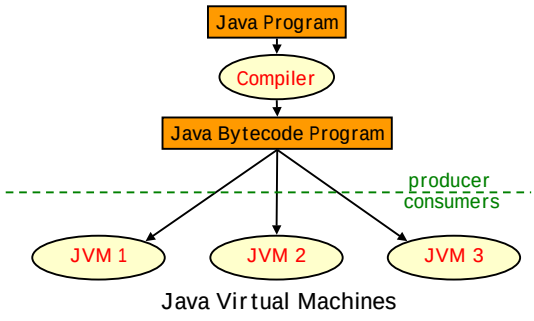
Problem: Too Many Platforms!

- Operating systems
  - DOS, Win95, 98, NT, ME, 2K, XP, Vista, ...
  - Unix, Linux, FreeBSD, Aix, ...
  - VM/CMS, OS/2, Solaris, Mac OS X, ...
- Architectures
  - Pentium, PowerPC, Alpha, SPARC, MIPS, ...

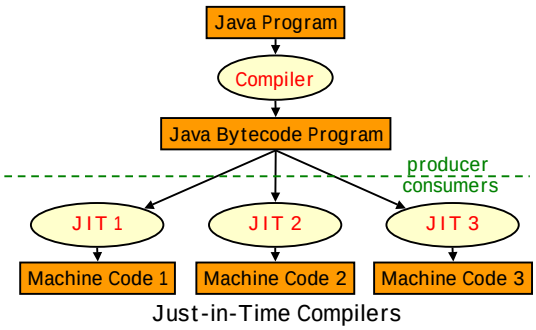
Dream: Platform Independence

- Compiler produces *one* low-level program for *all* platforms
  - Executed on a virtual machine (VM)
  - A different VM implementation needed for each platform, but installed once and for all

Platform Independence with Java



Platform Independence with Java

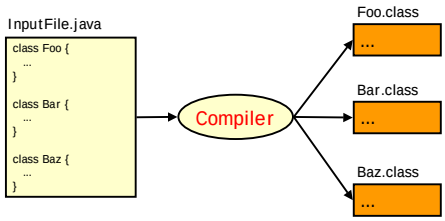


Just-in-Time Compilers

Java Bytecode

- Low-level compiled form of Java
- Platform-independent
- Compact
  - Suitable for mobile code, applets
- Easy to interpret
  - Java virtual machine (JVM) in your browser
  - Simple stack-based semantics
  - Support for objects

Class Files



What's in a Class File?

- Magic number, version info
- Constant pool
- Super class
- Access flags (public, private, ...)
- Interfaces
- Fields
  - Name and type
  - Access flags (public, private, static, ...)
- Methods
  - Name and signature (argument and return types)
  - Access flags (public, private, static, ...)
  - Bytecode
  - Exception tables
- Other stuff (source file, line number table, ...)

Class File Format

magic number	4 bytes	**0xCAFEBABE**
major version	2 bytes	**0x0021**
minor version	2 bytes	**0x0000**

- Magic number identifies the file as a Java class file
- Version numbers inform the JVM whether it is able to execute the code in the file

Constant Pool

CP length	2 bytes
CP entry 1	(variable)
CP entry 2	(variable)
...	...

- Constant pool consists of up to 65536 = 2^{16} entries
- Entries can be of various types, thus of variable length

Constant Pool Entries

Utf8 (unicode)	literal string (2 bytes for length, characters)
Integer	Java int (4 bytes)
Float	Java float (4 bytes)
Long	Java long (8 bytes)
Double	Java double (8 bytes)
Class	class name
String	String constant -- index of a Utf8 entry
Fieldref	field reference -- name and type, class
Methodref	method reference -- name and type, class
InterfaceMethodref	interface method reference
NameAndType	Name and Type of a field or method

Example

```
class Foo {
  public static void main(String[] args) {
    System.out.println("Hello world");
  }
}
```

Q) How many entries in the constant pool?

A) 33

```
1)CONSTANT_Methodref[10](class_index = 6, name_and_type_index = 20)
2)CONSTANT_Fieldref[9](class_index = 21, name_and_type_index = 22)
3)CONSTANT_String[8](string_index = 23)
4)CONSTANT_Methodref[10](class_index = 24, name_and_type_index = 25)
5)CONSTANT_Class[7](name_index = 26)
6)CONSTANT_Class[7](name_index = 27)
7)CONSTANT_Utf8[1]("<init>")
8)CONSTANT_Utf8[1]("()V")
9)CONSTANT_Utf8[1]("Code")
10)CONSTANT_Utf8[1]("LineNumberTable")
11)CONSTANT_Utf8[1]("LocalVariableTable")
12)CONSTANT_Utf8[1]("this")
13)CONSTANT_Utf8[1]("LFoo;")
14)CONSTANT_Utf8[1]("main")
15)CONSTANT_Utf8[1]("([Ljava/lang/String;)V")
16)CONSTANT_Utf8[1]("args")
17)CONSTANT_Utf8[1]("[Ljava/lang/String;")
18)CONSTANT_Utf8[1]("SourceFile")
19)CONSTANT_Utf8[1]("Foo.java")
20)CONSTANT_NameAndType[12](name_index = 7, signature_index = 8)
21)CONSTANT_Class[7](name_index = 28)
22)CONSTANT_NameAndType[12](name_index = 29, signature_index = 30)
23)CONSTANT_Utf8[1]("Hello world")
24)CONSTANT_Class[7](name_index = 31)
25)CONSTANT_NameAndType[12](name_index = 32, signature_index = 33)
26)CONSTANT_Utf8[1]("Foo")
27)CONSTANT_Utf8[1]("java/lang/Object")
28)CONSTANT_Utf8[1]("java/lang/System")
29)CONSTANT_Utf8[1]("out")
30)CONSTANT_Utf8[1]("Ljava/io/PrintStream;")
31)CONSTANT_Utf8[1]("java/io/PrintStream")
32)CONSTANT_Utf8[1]("println")
33)CONSTANT_Utf8[1]("(Ljava/lang/String;)V")
```

Code Attribute of a Method

maxStack	2 bytes	max operand stack depth
maxLocals	2 bytes	number of local variables
codeLength	2 bytes	length of bytecode array
code	codeLength	the executable bytecode
excTableLength	2 bytes	number of exception handlers
exceptionTable	excTableLength	exception handler info
attributesCount	2 bytes	number of attributes
attributes	variable	e.g. LineNumberTable

Example Bytecode

```
if (b) x = y + 1;
  else x = z;
```

```
5:  iload_1          //load b
6:  ifeq 16          //if false, goto else
9:  iload_3          //load y
10: iconst_1         //load 1
11: iadd             //y+1
12: istore_2         //save x
13: goto 19          //skip else
16: iload_4          //load z
18: istore_2         //save x
19: ...
```

then clause

else clause

Examples

```
class Foo {
   public static void main(String[] args) {
      System.out.println("Hello world");
   }
}
```

Q) How many methods?

A) 2


public static void main (String[] args)
Code: maxStack=2 maxLocals=1 length=9
exceptions=0
attributes=2
source lines=2
local variables=1
java/lang/String[] args startPC=0 length=9 index=0
0: getstatic java/lang/System.out
3: ldc "Hello world"
5: invokevirtual java/io/PrintStream.println(Ljava/lang/String;)V
8: return
void <init> ()
Code: maxStack=1 maxLocals=1 length=5
exceptions=0
attributes=2
source lines=1
local variables=1
Foo this startPC=0 length=5 index=0
0: aload_0
1: invokespecial java/lang/Object.<init>()V
4: return