Agenda for Today

- Object representation and layout
- Field access
- What is this?
- Object creation - new
- Method calls
  - Dynamic dispatch
  - Method tables
  - Super
- Runtime type information

OverLoading and Hiding

class One {
    int tag;
    int it;
    void setTag() { tag = 1; }
    int getTag() { return tag; }
    void setIt(int it) { this.it = it; }
    int getIt() { return it; }
}
class Two extends One {
    int it;
    void setTag() {
        tag = 2; it = 3;
    }
    int getThat() { return it; }
    void resetIt() {
        super.setIt(42);
    }
}
What does this program print?

class One {
    int tag;
    int it;
    void setTag()    { tag = 1; }
    int getTag()      { return tag; }
    void setIt(int it) {this.it = it; }
    int getIt()         { return it; }
}

class Two extends One {
    int it;
    void setTag() {
        tag = 2;  it = 3;
    }
    int getThat() { return it; }
    void resetIt() { super.setIt(42); }
}

public static void main(String[] args) {
    Two two = new Two();
    One one = two;
    one.setTag();
    System.out.println(one.getTag());
    one.setIt(17);
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}

Your Answer Here

Object Representation

- The naïve explanation is that an object contains
  - Fields declared in its class and in all superclasses
    - Redeclaration of a field hides superclass instance
  - Methods declared in its class and in all superclasses
    - Redeclaration of a method overrides (replaces) But overridden methods can still be accessed by super...
- When a method is called, the method inside that particular object is called
  - But we don't want to really implement it this way – we only want one copy of each method's code
Actual representation

- Each object contains
  - An entry for each field (variable)
  - A pointer to a runtime data structure describing the class
    - Key component: method dispatch table
- Basically a C/C++ struct
- Fields hidden by declarations in extended classes are still allocated in the object and are accessible from superclass methods

Method Dispatch Tables

- Often known as “vtables”
- One pointer per method
- Offsets fixed at compile time
- One instance of this per class, not per object

Method Tables and Inheritance

- Simple implementation
  - Method table for extended class has pointers to methods declared in it
  - Method table also contains a pointer to parent class method table
  - Method dispatch
    - Look in current table and use it if local
    - Look in parent class table if not local
    - Repeat
  - Actually used in some dynamic systems
O(1) Method Dispatch

- Idea: First part of method table for extended class has pointers in same order as parent class
  - BUT pointers actually refer to overriding methods
  - Method dispatch is indirect using fixed offsets known at compile time – O(1)
    - In C: *(object->vtbl[offset])(parameters)
- Pointers to additional methods in extended class are included following inherited/overridden ones

Method Dispatch Footnotes

- Still want pointer to parent class method table for other purposes
  - Casts and instanceof
  - Multiple inheritance requires more complex mechanisms
  - Also multiple interfaces in perverse situations(!)
Perverse Example Revisited

class One {
    int tag;
    int it;
    void setTag() { tag = 1; }
    int getTag() { return tag; }
    int getIt() { return it; }
}

class Two extends One {
    int it;
    void setTag() {
        tag = 2; it = 3;
    }
    int getThat() { return it; }
    void resetIt() { super.setIt(42); }
}

public static void main(String[] args) {
    Two two = new Two();
    One one = two;
    one.setTag();
    System.out.println(one.getTag());
    one.setIt(17);
    two.setTag();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
    two.resetIt();
    System.out.println(two.getIt());
    System.out.println(two.getThat());
}

Implementation & Trace

Now What?
- Need to explore
  - Object layout in memory
  - Compiling field references
    - Implicit and explicit use of "this"
  - Representation of vtables
  - Object creation – new
  - Code for dynamic dispatch
    - Including implementing "super.f"
  - Runtime type information – instanceof and casts
**Object Layout**
- Typically, allocate fields sequentially
- Follow processor/OS alignment conventions if appropriate
- Use first 32 bits of object for pointer to method table
- Objects are allocated on the heap
  - No actual representation in the generated code

**Local Variable Field Access**
- Source
  ```
  int n = obj.fld;
  ```
- X86
  ```
  mov eax,[ebp+offset obj] ; load obj
  mov eax,[eax+offset fld] ; load fld
  mov [ebp+offset n],eax ; store n
  ```

**Local Fields**
- A method can refer to fields in the receiving object either explicitly as "this.f" or implicitly as "f"
  - Both compile to the same code – an implicit "this." is inserted if not present
- Mechanism: a reference to the current object is an implicit parameter to every method
  - Can be in a register or on the stack
Source Level View

- When you write
  
  ```
  void setIt(int it) {
      this.it = it;
  }
  ```

- You really get
  
  ```
  void setIt(ObjType this, int it) {
      this.it = it;
  }
  ```

```
  obj.setIt(42);
  ```

x86 Conventions

- `ecx` is traditionally used as "this"
- Add to method call
  
  ```
  mov ecx, receivingObject; ptr to object
  ```

- Do this after arguments are evaluated and pushed, right before dynamic dispatch code (more about that to come)
- Need to save `ecx` in a temporary in methods that call other non-static methods
  - One possibility: add to prologue
  - Following examples aren't careful about this

x86 Local Field Access

- Source
  
  ```
  int n = fid; or int n = this.fid;
  ```

- X86
  
  ```
  mov eax, [ecx+ offset fid]; load fid
  mov [ebp+ offset_]eax; store n
  ```
x86 Method Tables (vtbls)

- These are generated in the assembly language source program.
- Need to pick a naming convention for method labels; suggestion:
  - For methods, classname$methodname
  - Need something more sophisticated to implement overloading
  - For the vtables themselves, classname$${}$
- First method table entry points to superclass table
- Also useful: second entry points to constructor
  - Makes implementation of super() particularly simple

Method Tables For Perverse Example

class One {
  void setTag() { … }
  int getTag() { … }
  void setIt(int it) { … }
  int getIt() { … }
}
class Two extends One {
  void setTag() { … }
  int getTag() { … }
  void resetIt() { … }
}

.data
One$$ dd 0 ; no superclass
d One$One
d One$setTag
d One$getTag
d One$setIt
d One$getIt

Two$$ dd One$$ ; parent
d Two$One$getIt
Twick$setIt
d Two$resetIt

Method Table Footnotes

- Key point: First four non-constructor method entries in Two’s method table are pointers to methods declared in One in exactly the same order
  - Compiler knows correct offset for a particular method regardless of whether that method is overridden
Object Creation – new

Steps needed
- Call storage manager (malloc or similar) to get the raw bits
- Store pointer to method table in the first 4 bytes of the object
- Call the constructor
- Result of new is pointer to the constructed object

Object Creation

Source
One one = new One(...);

X86
push nBytesNeeded ; obj size + 4
mov esp,4
lea esp,One$$ ; get method table address
mov [esp],edx ; store at beginning of object
mov ecx,eax ; set up "this" for constructor
<push constructor arguments> ; arguments (if needed)
call One$One ; call constructor
<pop constructor arguments> ; (if needed)
pop eax ; recover ptr to object
mov [ebp+offsetone],eax ; store n

Constructor

Only special issue here is generating call to superclass constructor
- Same issues as super.method(...) calls – defer for now
Method Calls

- Steps needed
  - Push arguments as usual
  - Put pointer to object in ecx (this)
  - Get pointer to method table from first 4 bytes of object
  - Jump indirectly through method table
  - Restore ecx to point to current object (if needed)
  - Useful hack: push it in the function prologue so it is always on the stack in a known location

Method Call

- Source
  - obj.meth(...);

- X86
  - <push arguments from right to left> ; (if needed)
  - mov ecx,[ebp+offset obj] ; get pointer to object
  - mov eax,[ecx] ; get pointer to method table
  - call dword ptr [eax+offsetmeth] ; call indirect via method tbl
  - <pop arguments> ; (if needed)
  - mov ecx,[ebp+offsetecxtemp] ; (if needed)

Handling super

- Almost the same as a regular method call with one extra level of indirection
- Source
  - super.meth(...);

- X86
  - <push arguments from right to left> ; (if needed)
  - mov ecx,[ebp+offset super] ; get pointer to object
Runtime Type Checking

- Use the method table for the class as a “runtime representation” of the class
- The test for “o instanceof C” is
  - Is o’s method table pointer == &C$?
  - Recursively, get the superclass’s method table pointer from the method table and check that
  - Stop when you reach Object (or a null pointer, depending on how you represent things)

Coming Attractions

- Code Generation
- Two models
  - Simple tree walk, which is adequate to complete the project
  - More standard instruction selection/scheduling/register allocation regime
- Rest of the quarter – survey of optimization plus special topics