Implementing Objects

Lecture 10
CS 212 - Fall 2007

Variables within Classes

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

Code for getting the value of a field
- \( \text{PUSHOFF fbrOffsetOfObjectRef} // \text{Push address of object} \)
- \( \text{PUSHIMM offsetOfField} // \text{Push fields offset} \)
- \( \text{ADD} // \text{Absolute address of field} \)
- \( \text{PUSHIND} // \text{Push value stored at that address} \)

Code for setting the value of a field
- \( \text{PUSHOFF fbrOffsetOfObjectRef} // \text{Push address of object} \)
- \( \text{PUSHIMM offsetOfField} // \text{Push fields offset} \)
- \( \text{ADD} // \text{Absolute address of field} \)
- \( \text{PUSHIMM valueToStore} // \text{Value to place into field} \)
- \( \text{STOREIND} // \text{Store value into address} \)

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 \( \text{a.sum()} \) then the extra parameter is \( \text{a} \).
  (Actually, the address of \( \text{a} \))

Function Call vs. Method Call

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

Calling a Constructor

- Goal: On return, address of new object should be on top of stack.
- In Bali, we have the extra goal that the constructor should be usable as regular function.

- Basically, a constructor is just a function.
- Build a standard stack frame.
- Called as a constructor.
  The return value is the new object—it is created when the constructor is called.
- Called as a function.
  The return value is the extra method parameter.
Function Call vs. Constructor 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/create object (need size)
  - Push arguments
  - Push/update FBR
  - Push/update PC
- Callee:
  - Push local variables
  - Execute constructor code
  - Clear local variables
  - Pop/restore PC
  - Caller:
    - Pop/restore FBR
    - Clear arguments
    - (Ret value is left on stack)

Shared Data for a Class
- Instances of the same class share the same dispatch vector
- This implies that your sam-code must create a dispatch vector 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
  - We don’t have static variables in Bali
- For each class:
  - Same dispatch vector (with any new stuff at the end)
  - Same object layout (with any new stuff at the end)
  - Or an instance of some subclass
  - An instance of the class itself
  - But the private fields and methods still exist
  - They refer to exactly the same set of methods
  - For a field
    - Address of object
    - Offset 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
  - 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

Dispatch Vector
- For a method, we don’t store a copy of the method’s code with each class instance
  - Instead we 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
  - Executing the code for a method
    - Push arguments
    - Push/create object (need size)
    - Push/update FBR
    - Push/update PC
    - Push/restore FBR
    - Push/restore PC

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 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
    - Parameter types & return type
    - Offset within dispatch vector
  - Private fields and methods can be removed from table after class has been compiled
    - Note all Bali fields and methods are public
  - 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
    - Some dispatch vector (with any new stuff at the end)
    - Some object layout (with any new stuff at the end)
  - This allows a method’s code to still work even though it’s dealing with subclasses
  - 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 is 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 is 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 (and extended-Bali) allow 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