Week 11
Implementing Objects

Paul Chew
CS 212 – Spring 2004

Announcements

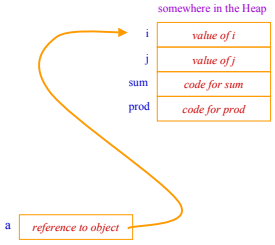
- Part 3
  - Be sure you handle *all* parts of the Part 3 grammar!
  - There is a .jar file containing
    - BaliSemanticException.java
    - BaliSyntaxException.java
    - Compiler.java
    - IllegalBaliException.java
    - MultipleBaliException.java
  - Use this .jar file instead of individual files
- To use the Part 3 .jar file
  - Make sure the .jar file is on your class path (in DrJava, look under Edit:Preferences)
  - Place the following code at start of any file that needs to use stuff from the .jar file:

```
import edu.cornell.cs.cs212.sp2004.part3.*;
```

- Sections are meeting
  - Today
  - Next Monday, too
- Make use of Office Hours!

Intuitive View of an Object

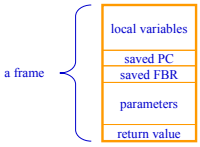
```
class A {
   int i, j;

   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

Calling a Constructor

- Goal: On return, address of new object should be on top of stack
- Basically, a constructor is just a function
  - Build a standard stack frame
  - Include one extra parameter: the newly created object



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 space for ret value
  - Push/create object (need size)
  - Push arguments
  - Push/update FBR
  - Push/update PC
- Callee:
  - Push local variables
  - Execute constructor code
  - Copy object ref to ret value
  - Clear local variables
  - Pop/restore PC
- Caller:
  - Pop/restore FBR
  - Clear arguments
  - (Ret value is left on stack)

Variables

- Local variables 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

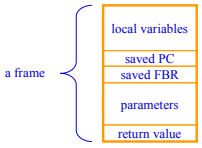
```
PUSHOFF offsetOfObjectRef     // Push address of object
PUSHIMM offsetOfField         // Push field's offset
ADD                           // Absolute address of field
PUSHIND                       // Push value stored at that address
```

- Code for *setting* the value of a field

```
PUSHOFF offsetOfObjectRef     // Push address of object
PUSHIMM offsetOfField         // Push field's offset
ADD                           // Absolute address of field
PUSHIMM valueToStore          // Value to place into field
STOREIND                      // 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
    `a.sum()`
    then the extra parameter is **a** (actually, the *address of a*)



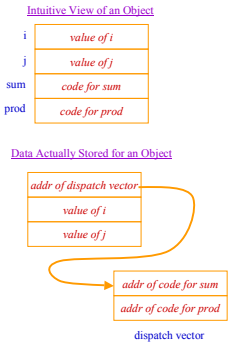
Function Call vs. Method Call

- 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 address
  - (Ret value is left on stack)

An Object's Methods

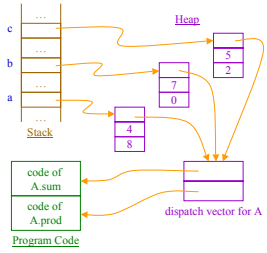
- For a method defined within a class, we *don't* store a copy of the method's code with *each* class instance
  - Instead we can store the *address* of the method's code
- But each instance of a class will refer to exactly the same set of methods
  - Thus, it's wasteful for each object to store an address 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 your sam-code must create a dispatch vector (in the Heap) 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

```
a = new A(4, 8);
b = new A(7, 0);
c = new A(5, 2);
```



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 type 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 method
    - Examples: i or sum() when used within a method of A

Multiple Symbol Tables

- *Program Symbol Table*
  - Classes
    - Where to find class's dispatch vector
    - Size of corresponding object
  - Functions & constructors
    - Signature
  - May want to build during separate pass over the AST

- *Class Symbol Table*
  - Fields
    - Type & offset within object
  - Methods
    - Signature & 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

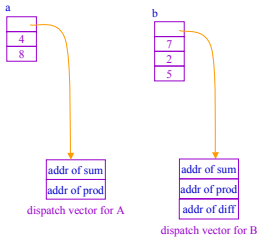
Inheritance

- 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 defines 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, code is generated based on the object's *declared* type
  - This is done by appending the 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 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