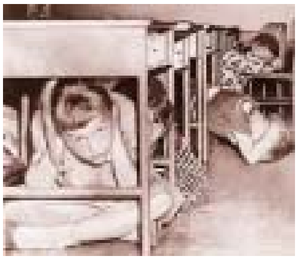

A class in the 1950s
"Duck & Cover"

Compiling Bali Part 4: Classes and Arrays

Lecture 11
CS 212 – Fall 2007

New in Bali4

- Classes

program -> [declarations] :
    (class | function)*
class -> class *name* :
    [declarations] :
    function*
    endclass
type -> (int | boolean | void |
    name) ([ ])
fieldRef -> . *name*

- Arrays

reference -> (*name* | this) modifier*
modifier -> subscript | functionArgs |
    fieldRef
subscript -> [expression]
term -> *literal* | (expression) |
    arrayValue | inputValue |
    reference
arrayValue -> type arrayElementList
arrayElementList -> { [expression
    (, expression)*] }

Example Bali4 Code

```
# Sample Part 4 program that uses
# a Queue and a Stack
:
int main ( ) :
  int n, Stack s, Queue q :
  n = 0; s = Stack(); q = Queue();
  loop while n < 5;
    s.put(n); q.put(n);
    n = n + 1;
  endloop
  n = 0;
  loop while n < 5;
    print s.get(), q.get();
    n = n + 1;
  endloop
  return 0;
end
```

```
class Node :
  int data, Node link :

  # Constructor
  Node Node (int data, Node link) ::
    this.data = data;
    this.link = link;
  end

endclass
```

Example Bali4 Code, Continued

```
class Queue :
  Node head, Node last :

  void put (int i) : Node n :
    n = Node(i, n);
    if head == null then head = n;
    else last.link = n;
    endif
    last = n;
    return;
  end

  int get () : Node n :
    n = head;
    head = head.link;
    return n.data;
  end
endclass
```

```
class Stack :
  Node top :

  void put (int i) : :
    top = Node(i, top);
    return;
  end

  int get () :
    Node n :
    n = top;
    top = top.link;
    return n.data;
  end
endclass
```

Rules for Classes

- No inheritance
  - But you can do inheritance as bonus work
- Fields and methods are all local to the class's namespace
  - They are accessible from within constructors and methods in the same class
  - Can use *this.fieldName* when a field and a local variable have same name
- All fields and methods of a class are public
- A field and a method cannot share the same name (this is different from Java)

Rules for Constructors

- There is no *new* keyword
  - A constructor is called like a function with class-name used in place of a function-name
- Within a constructor, the only semantically valid return-statement is the version with no expression
  - It acts as if it is *return this;*
- Within a class, the constructor is simply a method with the class-name used as both the return-type and the function name
  - If no constructor is provided, an empty, default constructor (with no parameters) is used

Rules for "this"

- As in Java, *this* refers to the current class instance
  - It is only valid within a method or constructor
- The form *this(arguments)* calls the class's constructor on the current instance
  - Note that the constructor is called, but no new instance is created; in effect it re-initializes the current instance
- The form *this.name* refers to a field of the current instance
- The form *this.name(arguments)* refers to a method of the current instance
- The form *this* (by itself) refers to the current instance

Rules for Arrays

- When an array is declared, its initial value is null
  - int[ ] values, # values == null
- The expression type[size] creates an array of the given size
  - values = int[9];
- It's also possible to create an array by listing its elements
  - values = int{7,0,5,2,4,6,3,8,1};
- Each array has a "field" called *size* representing the declared size of the array
  - if values.size > 4 then …
- Arrays of class instances are legal
  - Node[] nodeArray,
- As in Java, array subscripts are checked at *runtime*
  - Thus, every time you generate code for a subscript, you must generate code to check array bounds

Code to Create an Array

- For an array of size 9
  - You need one extra word for the size-field

```
PUSHIMM 1 + numOfElements
MALLOC      // Get heap block
DUP
PUSHIMM numOfElements
STOREIND    // Set size-field
Store array's address
```

- You may want to store (array's address + 1) since this is where the elements start

What Info is Needed to Generate Code?

- For a local variable
  - Offset from FBR
- For a field
  - Offset of field from start of object
- For a global variable
  - Absolute location of variable
- For a method
  - Offset of method from start of dispatch vector
- For a constructor
  - The size (# of fields) of the object
  - Location of the dispatch vector for the class
- You will need more than one pass over the AST because you cannot generate code until you know
  - The return type and the parameter types for each function
    - Need this to type-check and to generate code for a function call
  - Size of each object
    - Need this to create code for a constructor call
  - Each method's offset in dispatch vector
    - Need this to create code for a method call

Outline for Bali4 Compiler

- Build the AST
- Walk the AST to determine
  - Function info
  - Size (# fields) for each class
    - A class can inherit fields
  - Dispatch vector for each class
    - A class can inherit a dispatch vector
- Create code for each class's dispatch vector
- Walk the AST again, generating code for functions, constructors, and methods
- This is just one possible way to compile Bali4
  - You don't have to use this outline
    - For example, since we don't have inheritance, you don't *have to* use a dispatch vector
- For our example
  - Node
    - Size = 2; DV = empty
  - Queue
    - Size = 2; DV = put, get
  - Stack
    - Size = 1; DV = put, get

Dispatch Vectors

- The simplest method is to build each dispatch vector as part of the program code

```
"DV$Queue":
   JUMP "M$Queue$put"
   JUMP "M$Queue$get"
"DV$Stack":
   JUMP "M$Stack$put"
   JUMP "M$Stack$get"
```

- Idea is that the object itself stores the location of its dispatch vector
  - Example: a Queue object stores the address "DV$Queue"
- Code to call a method
  - Push space for ret value
  - Push any arguments
  - Push the address of object's DV
  - Push the method's offset
  - Add (offset to addressOfDV)
  - Push/update FBR (LINK)
  - Push/update PC (JSRIND)
  - Restore FBR (UNLINK)
  - Clear arguments from stack

Initial Program Code

```
program:
   PUSHIMM 0     // For exit code
   Reserve space for global variables
   PUSHIMM 0     // Main's ret value
   LINK          // New stack frame
   JSR "F$main"  // Jump to main func
   UNLINK        // Restore FBR
   STOREABS 0    // Set exit code
   Clear global variables
   STOP
"DV$Queue":
   JUMP "M$Queue$put"
   JUMP "M$Queue$get"
"DV$Stack":
   JUMP "M$Stack$put"
   JUMP "M$Stack$get"
```

- Recall that when a class *inherits* from another class
  - It uses the same dispatch vector (with any new stuff on the end)
  - This is necessary so that an instance of the class works correctly when using methods of its super class

Runtime Errors

- Divide by zero
  - You don't have to do anything
- Array index out-of-bounds
  - Clear the stack, place an error code at position 0, and stop
  - Error code = -1
- Use of a null pointer
  - Clear the stack, place an error code at position 0 and stop
  - Error code = -2
- You can design your own sam-code subroutines
  - For example
    - To check array index out-of-bounds
    - To check for null pointer
    - To clear stack, place an error code, and stop
  - The code for these subroutines can be generated as part of your initial program code

Code for main

```
int main ( ) :
  int n, Stack s, Queue q :
  n = 0; s = Stack(); q = Queue();
  loop while n < 5;
    s.put(n); q.put(n);
    n = n + 1;
  endloop
  n = 0;
  loop while n < 5;
    print s.get(), q.get();
    n = n + 1;
  endloop
  return 0;
end
```

```
"F$main":
PUSHIMM 0          // n, offset = 2
PUSHIMM 0          // s, offset = 3
PUSHIMM 0          // q, offset = 4
PUSHIMM 0
STOREOFF 2         // n=0
PUSHIMM 3          // Space for Stack
MALLOC             // Create a Stack
DUP                // Addr of Stack
PUSHIMMPA "DV$Stack"
STOREIND           // Store DV address
STOREOFF 3         // Store stack in s
Similar code for q = Queue()
...
```

Code for "q.put(n)"

- The calling code for a method looks like this:
  - <Code to place space for return value on stack>
  - <Code to place arguments on Stack>
  - <Code to save/update FBR>
  - <Code to place address of method on top of stack>
  - JSRIND
  - <Code to restore FBR>
  - <Code to clear arguments from Stack>
- Recall: q is treated as an additional (implicit) argument

```
PUSHIMM 0     // No return value
PUSHOFF 4     // q
PUSHOFF 2     // n
LINK          // Store/update FBR
PUSHIMM 0     // Offset for put method
PUSHOFF -2    // Address of object (q)
PUSHIND       // Addr of dispatch vector
ADD           // Addr of correct method
JSRIND        // Method call
UNLINK        // Restore FBR
ADDSP -2      // Clear arguments
```

Code for "print q.get()"

- The calling code for a method looks like this:
  - <Code to place space for return value on Stack>
  - <Code to place arguments on Stack>
  - <Code to save/update FBR>
  - <Code to place address of method on top of stack>
  - JSRIND
  - <Code to restore FBR>
  - <Code to clear arguments from Stack>
- Recall: q is treated as an additional (implicit) argument

```
PUSHOFF 0     // Space for RV
PUSHOFF 4     // q
LINK          // Store/update FBR
PUSHIMM 1     // Offset for get method
PUSHOFF -1    // Addr of object
PUSHIND       // Addr of dispatch vector
ADD           // Addr of correct method
JSRIND        // Method call
UNLINK        // Restore FBR
ADDSP -1      // Clear arguments
WRITE         // Print the result
```

Code for a Constructor (Node)

- For this example, there are no local variables
- There are 3 arguments: the (implicit) object and the two explicit arguments

```
class Node :
  int data, Node link :

  # Constructor
  Node Node (int data, Node link) ::
    this.data = data;
    this.link = link;
  end

endclass
```

- Recall: The calling code allocates the space and passes the new object (along with any other arguments)

```
"C$Node$":
PUSHOFF -3    // Push addr of 'this'
PUSHIMM 1     // Offset for this.data
ADD           // Addr of this.data
PUSHOFF -2    // Push data (the arg)
STOREIND      // Store into this.data
PUSHOFF -3    // Push addr of 'this'
PUSHIMM 2     // Offset for this.link
ADD           // Address of this.link
PUSHOFF -1    // Push link (the arg)
STOREIND      // Store into this.link
JUMPIND       // Return
```

Code for “n = Node(i, n)”

- The calling code for a constructor looks like this

<Push/create object (need size); use as ret value>
<Push arguments>
<Push/update FBR>
<Push/update PC (i.e., jump to constructor)>
<Pop/restore FBR>
<Clear arguments (ret value is left on stack)>

```
PUSHIMM 3          // Size of Node
MALLOC             // Constr for Node
PUSHOFF -1         // Push argument i
PUSHOFF 2          // Push argument n
LINK               // Save/update FBR
JSR "C$Node"
UNLINK             // Restore FBR
ADDSP -2           // Clear args (not rv)
STOREOFF 2         // Store into n
```