CS61B Lecture #5: Arrays and Objects

- Homeworks are generally due by the next lab.
- For next week, please read *Head First Java*, chapters 5 and 6.
- **Discussion Change:** Starting next Thursday (13 September), discussion section 111 (10-11AM) will move from 3109 Etch. to 6 Evans.
Arrays

• An array is a structured container whose components are
  - **length**, a fixed integer.
  - a sequence of **length** simple containers of the same type, numbered from 0.
  - (.length field usually implicit in diagrams.)

• Arrays are anonymous, like other structured containers.

• Always referred to with pointers.

• For array pointed to by A,
  - Length is \( A.length \)
  - Numbered component \( i \) is \( A[i] \) (\( i \) is the index)
  - Important feature: index can be **any integer expression**.
A Few Samples

Java

```java
int[] x, y, z;
String[] a;
x = new int[3];
y = x;
a = new String[3];
x[1] = 2;
y[1] = 3;
a[1] = "Hello";

int[] q;
qu = new int[] { 1, 2, 3 };
// Short form for declarations:
int[] r = { 7, 8, 9 };
```

Results

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>r</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

Hello
Example: Accumulate Values

Problem: Sum up the elements of array A.

static int sum (int[] A) {
    int N;
    N = 0;
    for (int i = 0; i < A.length; i += 1)
        N += A[i];
    return N;
}

// For the hard-core: could have written

int N, i;
for (i=0, N=0; i<A.length; N += A[i], i += 1)
    { } // or just ;

// But please don’t: it’s obscure.

Last modified: Mon Oct 22 15:34:04 2007
**Example: Insert into an Array**

**Problem:** Want a call like `insert(A, 2, "gnu")` to convert (destructively)

A: bear

```
  |  |  |  |  |
  v  v  v  v  v
bear   gazelle  hartebeest  skunk
```

A: bear

```
  |  |  |  |  |
  v  v  v  v  v
gazelle  gnu  hartebeest
```

`/** Insert X at location K in ARR, moving items
 * K, K+1, ... to locations K+1, K+2, ....
 * The last item in ARR is lost. */`

```
static void insert (String[] arr, int k, String x) {
    for (int i = arr.length-1; i > k; i -= 1) // Why backwards?
        arr[i] = arr[i-1];
    // Alternative to this loop:
    // System.arraycopy ( arr, k, arr, k+1, arr.length-k-1);
    arr[k] = x;
}
```
**Growing an Array**

**Problem:** Suppose that we want to change the description above, so that \( A = insert2 (A, 2, \text{"gnu"}) \) does not shove “skunk” off the end, but instead “grows” the array.

```java
/** Return array, r, where r.length = ARR.length+1; r[0..K-1]
 * the same as ARR[0..K-1], r[k] = x, r[K+1..] same as ARR[K..]. */
static String[] insert2 (String[] arr, int k, String x) {
    String[] result = new String[arr.length + 1];
    System.arraycopy (arr, 0, result, 0, k);
    System.arraycopy (arr, k, result, k+1, arr.length-k);
    result[k] = x;
    return result;
}
```

- Why do we need a different return type from `insert`??
Object-Based Programming

Basic Idea.

- **Function-based programs** are organized primarily around the functions (methods, etc.) that do things. Data structures (objects) are considered separate.

- **Object-based programs** are organized around the types of objects that are used to represent data; methods are grouped by type of object.

- Simple banking-system example:

  Function-based

  - account
    - deposit
    - withdraw
    - account

  Object-based

  - Account
    - deposit
    - withdraw
    - balance: 1420
    - Exported methods
    - Exported field
Philosophy

- Idea (from 1970s and before): An abstract data type is
  - a set of possible values (a domain), plus
  - a set of operations on those values (or their containers).

- In IntList, for example, the domain was a set of pairs: (head, tail), where head is an int and tail is a pointer to an IntList.

- The IntList operations consisted only of assigning to and accessing the two fields (head and tail).

- In general, prefer a purely procedural interface, where the functions (methods) do everything—no outside access to fields.

- That way, implementor of a class and its methods has complete control over behavior of instances.

- In Java, the preferred way to write the “operations of a type” is as instance methods.
You Saw It All in CS61A: The Account class

(define-class (account balance0)
  (instance-vars (balance 0))
  (initialize
    (set! balance balance0))
  (method (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (method (withdraw amount)
    (if (< balance amount)
      (error "Insufficient funds")
      begin
        (set! balance (- balance amount))
        balance)))

(define my-account
  (instantiate account 1000))
(ask my-account 'balance)
(ask my-account 'deposit 100)
(ask my-account 'withdraw 500)

public class Account {
  public int balance;
  public Account (int balance0) {
    balance = balance0;
  }
  public int deposit (int amount) {
    balance += amount; return balance;
  }
  public int withdraw (int amount) {
    if (balance < amount)
      throw new IllegalStateException
        ("Insufficient funds");
    else balance -= amount;
    return balance;
  }
}

Account myAccount = new Account (1000);
myAccount.balance
myAccount.deposit (100);
myAccount.withdraw(500);
The Pieces

- Class declaration defines a new type of object, i.e., new type of structured container.
- **Instance variables** such as balance are the simple containers within these objects (**fields or components**).
- **Instance methods**, such as deposit and withdraw are like ordinary (static) methods that take an invisible extra parameter (called **this**).
- The **new** operator creates (**instantiates**) new objects, and initializes them using constructors.
- **Constructors** such as the method-like declaration of Account are special methods that are used only to initialize new instances. They take their arguments from the **new** expression.
- **Method selection** picks methods to call. For example,

  myAccount.deposit(100)

  tells us to call the method named deposit that is defined for the object pointed to by myAccount.
Getter Methods

• Slight problem with Java version of Account: anyone can assign to the balance field.

• This reduces the control that the implementor of Account has over possible values of the balance.

• Solution: allow public access only through methods:

```java
public class Account {
    private int balance;
    ...
    public int balance () { return balance; }
    ...
}
```

• Now the balance field cannot be directly referenced outside of Account.

• (OK to use name balance for both the field and the method. Java can tell which is meant by syntax: A.balance vs. A.balance().)
Class Variables and Methods

• Suppose we want to keep track of the bank’s total funds.
• This number is not associated with any particular Account, but is common to all—it is *class-wide*.
• In Java, “class-wide” ≡ static

```java
public class Account {
    ...
    private static int funds = 0;
    public int deposit (int amount) {
        balance += amount; funds += amount;
        return balance;
    }
    public static int funds () {
        return funds;
    }
    ...
    // Also change withdraw.
}
```

• From outside, can refer to either `Account.funds()` or `myAccount.funds()` (same thing).
Instance Methods

- Instance method such as

```java
int deposit (int amount) {
    balance += amount; funds += amount;
    return balance;
}
```

behaves sort of like a static method with hidden argument:

```java
static int deposit (final Account this, int amount) {
    this.balance += amount; funds += amount;
    return this.balance;
}
```

- NOTE: Just explanatory: Not real Java (not allowed to declare 'this'). (final is real Java; means “can’t change once set.”)

- Likewise, the instance-method call `myAccount.deposit (100)` is like a call on this fictional static method:

  ```java
  Account.deposit (myAccount, 100);
  ```

- Inside method, as a convenient abbreviation, can leave off leading 'this.' on field access or method call if not ambiguous.
'Instance' and 'Static' Don't Mix

• Since real static methods don't have the invisible \texttt{this} parameter, makes no sense to refer directly to instance variables in them:

  ```java
  public static int badBalance (Account A) {
      int x = A.balance;  // This is OK (A tells us whose balance)
      return balance;      // WRONG! NONSENSE!
  }
  ```

• Reference to \texttt{balance} here equivalent to \texttt{this.balance},

• But this is meaningless (\textit{whose} balance?)

• However, it makes perfect sense to access a static (class-wide) field or method in an instance method or constructor, as happened with \texttt{funds} in the \texttt{deposit} method.

• There's only one of each static field, so don't need to have a 'this' to get it. Can just name the class.
Constructors

• To completely control objects of some class, you must be able to set their initial contents.

• A constructor is a kind of special instance method that is called by the new operator right after it creates a new object, as if

\[
L = \text{new IntList}(1, \text{null}) \Rightarrow \begin{cases} 
\text{tmp} = \text{pointer to } 0; \\
\text{tmp}.\text{IntList}(1, \text{null}); \\
L = \text{tmp}; 
\end{cases}
\]

• Instance variables initializations are moved inside constructors:

```java
class Foo {
    int x = 5;
    Foo () {
        DoStuff ();
    }
    ...
}
```

```java
class Foo {
    int x;
    Foo () {
        x = 5;
        DoStuff ();
    }
    ...
}
```

• In absence of any explicit constructor, get default constructor:

public Foo() {
    ...
}

• Multiple overloaded constructors possible (different parameters).
## Summary: Java vs. CS61A OOP in Scheme

<table>
<thead>
<tr>
<th>Java</th>
<th>CS61A OOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>class Foo ...</td>
<td>(define-class (Foo args)...)</td>
</tr>
<tr>
<td>int x = ...;</td>
<td>(instance-vars (x ...))</td>
</tr>
<tr>
<td>Foo(args) {...}</td>
<td>(initialize ...)</td>
</tr>
<tr>
<td>int f(...) {...}</td>
<td>(method (f ...) ...)</td>
</tr>
<tr>
<td>static int y = ...;</td>
<td>(class-vars (y ...))</td>
</tr>
<tr>
<td>static void g(...) {...}</td>
<td>(define (g...)...)</td>
</tr>
<tr>
<td>aFoo.f (...)</td>
<td>(ask aFoo 'f ...)</td>
</tr>
<tr>
<td>aFoo.x</td>
<td>(ask aFoo 'x)</td>
</tr>
<tr>
<td>new Foo (...)</td>
<td>(instantiate Foo ...)</td>
</tr>
<tr>
<td>this</td>
<td>self</td>
</tr>
</tbody>
</table>