Basic Java Syntax and Control Flow

(and some algorithms as well)

Prof. Chris Jermaine
cmj4@cs.rice.edu

Basic Java Syntax and Control Flow

- Today may be a bit boring
- Will cover basic control flow and syntax in Java
  - For
  - While
  - Do-While
  - If
  - Method calls
  - Array accesses
  - Comments
  - Switch
  - more...
- Java borrows its basic syntax, control flow from C

Illustrative Example

- Best illustrated with an example: edit distance
  - — Studied this in Luay’s class, right?
  - — Given two strings *a*, *b*
  - — The “edit distance” is the number of “edit ops” needed to transform *a* into *b*
  - — An “edit op” is “insert”, “delete”, “xform”

Illustrative Example

- Best illustrated with an example: edit distance
  - — Studied this in Luay's class
  - — Given two strings *a, b*
  - — The "edit distance" is the number of "edit ops" needed to transform *a* into *b*
  - — An "edit op" is "insert", "delete", "xform"
- Example: *a* = 01101100, *b* = 110001000
  - — xform: *a* = **1**1101100, *b* = 110001000
  - — xform: *a* = 11**0**01100, *b* = 110001000
  - — xform: *a* = 1100**0**100, *b* = 110001000
  - — add: *a* = 11000100**0**, *b* = 110001000
  - — So edit distance is four

Edit Distance

- Classic algorithm (SW) based on following observation:
  - — ED(string1 + c, string2 + c) = ED (string1, string2)
  - — ED(string1 + c, string2) <= ED (string1, string2) + 1 [can always delete!]
  - — ED(string1, string2 + c) <= ED (string1, string2) + 1 [can always add!]
  - — ED(string1 + c1, string2 + c2) <= ED(string1, string2) + 1 [can always xform!]

Edit Distance

- Classic algorithm (SW) based on following observation:
  - — ED(string1 + c, string2 + c) = ED (string1, string2)
  - — ED(string1 + c, string2) <= ED (string1, string2) + 1 [can always delete!]
  - — ED(string1, string2 + c) <= ED (string1, string2) + 1 [can always add!]
  - — ED(string1 + c1, string2 + c2) <= ED(string1, string2) + 1 [can always xform!]
- So, let $A[i, j]$ be the fewest number of ops to obtain first j chars of b from the first i chars of a

Edit Distance

- Classic algorithm (SW) based on following observation:
  - — ED(string1 + c, string2 + c) = ED (string1, string2)
  - — ED(string1 + c, string2) <= ED (string1, string2) + 1 [can always delete!]
  - — ED(string1, string2 + c) <= ED (string1, string2) + 1 [can always add!]
  - — ED(string1 + c1, string2 + c2) <= ED(string1, string2) + 1 [can always xform!]
- So, let $A[i, j]$ be the fewest number of ops to obtain first j chars of b from the first i chars of a
- Then $A[i, j]$ is the minimum of:
  - — $A[i - 1, j - 1]$ (only available if $A[i] = A[j]$)
  - — $A[i - 1, j] + 1$
  - — $A[i, j - 1] + 1$
  - — $A[i - 1, j - 1] + 1$

To Code This Up in Java

- Assume a method

```
/**
 *  This method executes the Smith Waterman algorithm
 *  to find the edit distance between a and b
 */
public int editDistance (String a, String b) {

   // we first need our A array
   int[][] A = new int[a.length () + 1][b.length () + 1];
```

- What’s up with “new”?
  - — *int[][] A* declares a “reference” (like an address)
  - — But initially that address is empty
  - — *new* asks Java to allocate memory and return the resulting address
  - — If you don’t set *A* to something after decl, program will crash when you use it!

To Code This Up in Java

- Assume a method

```
/**
 *  This method executes the Smith Waterman algorithm
 *  to find the edit distance between a and b
 */
public int editDistance (String a, String b) {

   // we first need our A array
   int[][] A = new int[a.length () + 1][b.length () + 1];
```

- How’d I learn about the “length” method?
  - — Went to Google and typed “String java”
  - — 2nd result was “String (Java Platform SE 6)” at oracle.com
  - — Get used to having one or two browser windows open when coding!

Now Need To Initialize

```
public int editDistance (String a, String b) {

   // we first need our A array
   int[][] A = new int[a.length () + 1][b.length () + 1];

   // set A[i][0] to i, since to transform any string to ““
   // you just delete all of the characters
   for (int i = 0; i < a.length () + 1; i++) {
      A[i][0] = i;
   }
```

- Things to notice:
  - — Curly brackets *{* and *}* group statements together into a block
  - — *for* has three sub-statements (each can be almost **any** valid Java statement!)
  - — first is an intialization statement, run once
  - — second is a check run before each iteration (loop ends if evals to **false**)
  - — third is run after each iteration

Now Need To Initialize

```
public int editDistance (String a, String b) {

   // we first need our A array
   int[][] A = new int[a.length () + 1][b.length () + 1];

   // set A[i][0] to i, since to transform any string to ""
   // you just delete all of the characters
   for (int i = 0; i < a.length () + 1; i++) {
      A[i][0] = i;
   }
```

- Also note:
  - — I didn't need a *new* with *i* because it's of a built-in type (more in a lecture or two)
  - — This, *i* is an actual variable as opposed to a reference
  - — Array indices start at zero

Now Need To Initialize

```
public int editDistance (String a, String b) {

   // we first need our A array
   int[][] A = new int[a.length () + 1][b.length () + 1];

   // set A[i][0] to i, since to transform any string to ““
   // you just delete all of the characters
   for (int i = 0; i < a.length () + 1; i++) {
      A[i][0] = i;
   }
   // set A[0][j] to j, since to transform ““ to any string,
   // you just insert all of the characters
   for (int j = 0; j < b.length () + 1; j++) {
      A[0][j] = j;
   }
```

And Do the Actual Calculation

```
public int editDistance (String a, String b) {
    ...
    // now do the Smith-Waterman calculation
    for (int i = 1; i < a.length () + 1; i++) {
        for (int j = 1; j < b.length () + 1; j++) {
            if (a.charAt(i - 1) == b.charAt(j - 1)) {
                // if the two chars are the same, no edit op
                A[i][j] = A[i - 1][j - 1];
            } else {
                int best = A[i - 1][j]; // cost to delete
                if (A[i][j - 1] + 1 < best) // cost to insert
                    best = A[i][j - 1];
                if (A[i - 1][j - 1] < best) // cost xform
                    best = A[i - 1][j - 1];
                A[i][j] = best + 1; // record cost of best op
            }
        }
    }
```

Things To Notice

```
public int editDistance (String a, String b) {
    ...
    // now do the Smith-Waterman calculation
    for (int i = 1; i < a.length () + 1; i++) {
        for (int j = 1; j < b.length () + 1; j++) {
            if (a.charAt(i - 1) == b.charAt(j - 1)) {
                // if the two chars are the same, no edit op
                A[i][j] = A[i - 1][j - 1];
            } else {
                int best = A[i - 1][j]; // cost to delete
                if (A[i][j - 1] < best) // cost to insert
                    best = A[i][j - 1];
                if (A[i - 1][j - 1] < best) // cost to xform
                    best = A[i - 1][j - 1];
                A[i][j] = best + 1; // record cost of best op
            }
        }
    }
```

if accepts any statement that can eval to true/false

== checks for equality (be careful with =)

Things To Notice

```
public int editDistance (String a, String b) {
   ...
   // now do the Smith-Waterman calculation
   for (int i = 1; i < a.length () + 1; i++) {
      for (int j = 1; j < b.length () + 1; j++) {
         if (a.charAt(i - 1) == b.charAt(j - 1)) {
            // if the two chars are the same, no edit op
            A[i][j] = A[i - 1][j - 1];
         } else {
            int best = A[i - 1][j]; // cost to delete
            if (A[i][j - 1] < best) // cost to insert
               best = A[i][j - 1];
            if (A[i - 1][j - 1] < best) // cost to xform
               best = A[i - 1][j - 1];
            A[i][j] = best + 1; // record cost of best op
         }
      }
   }
```

Can declare a variable/reference anywhere...
must declare it before using it

Things To Notice

```
public int editDistance (String a, String b) {
    ...
    // now do the Smith-Waterman calculation
    for (int i = 1; i < a.length () + 1; i++) {
        for (int j = 1; j < b.length () + 1; j++) {
            if (a.charAt(i - 1) == b.charAt(j - 1)) {
                // if the two chars are the same, no edit op
                A[i][j] = A[i - 1][j - 1];
            } else {
                int best = A[i - 1][j]; // cost to delete
                if (A[i][j - 1] < best) // cost to insert
                    best = A[i][j - 1];
                if (A[i - 1][j - 1] < best) // cost to xform
                    best = A[i - 1][j - 1];
                A[i][j] = best + 1; // record cost of best op
            }
        }
    }
```

If block not explicitly specified using *{}*, block size is one stmnt (careful; only do it for one line stmnts!)

Returning a Value

```
public int editDistance (String a, String b) {
    ...
    // now do the Smith-Waterman calculation
    ...
    // and the final return value is in the bottom right
    // corner of A, since this is the cost to xfrom ALL of
    // a into ALL of b
    return A[a.length ()][b.length ()];
}
```

- Every non-null method needs to return a value!

A Note On Comments

- Notice how I’ve commented everyting
- Get used to documenting *as you code*
- The best programmers:
  - — Think carefully about what a block of code needs to do
  - — They describe this in English
  - — Then they write the code
  - — Get in the habit of doing it this way!

So Far…

- We have seen:
  - For, if, arrays, comments, new, code blocks, method calls, return statements
- What are we missing?
  - While loops
  - Ex: say we want to compute ceil of log (base b) of an int n
  - Simple alg: keep multiplying by b until you reach/exceed n

Writing a While Loop

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
   // accum is always b^pow
   int accum = 1, pow = 0;

   // repeatedly multiply accum by another b until
   // we hit n
   while (accum < n) {
      pow++;
      accum *= b;
   }
   return pow;
}
```

Stuff to Notice

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
   // accum is always b^pow
   int accum = 1, pow = 0;

   // repeatedly multiply accum by another b until
   // we hit n; accum is always b^pow
   while (accum < n) {
      pow++;
      accum *= b;
   }
   return pow;
}
```

Can declare, init. multiple vars/refs at once

This is the classic C notation for “eval. then increment”

C notation for “accum = accum * b”

Stuff to Notice

```
/**
 *  This method computes the log_b (n) in a silly way
 */
public int logBaseB (int b, int n) {
    // accum is always b^pow
    int accum = 1, pow = 0;

    // repeatedly multiply accum by another b until
    // we hit n; accum is always b^pow
    while (accum < n) {
        pow++;
        accum *= b;
    }
    return pow;
}
```

Can declare, init. multiple vars/refs at once

This is the classic C notation for “eval. then increment”

C notation for “accum = accum * b”

- Question: can we do better than linear in the return value?

Repeated Squaring

- Every number is the sum of some subset of $\{2^0, 2^1, 2^2, 2^3, 2^4...\}$
  - — Ex: $13 = 1 + 4 + 8$ (note you can compute this greedily!)
- So if $x = \log_b(n)$
  - — Then x is the sum of some subset of $\{2^0, 2^1, 2^2...\}$ 'cause x is a number!
- Say $x = 1 + 4 + 16$
  - — This (by definition of log) means $b^{(1 + 4 + 16)} = n$
  - — Equivalently, this means $b^1 b^4 b^{16} = n$
- So, all we have to do to compute the $\log_b(n)$...
  - — Is to find a set of powers of b that multiply together to get n
  - — And where each power is itself a power of 2
  - — In other words, form n by multiplying items from the set $\{b^1, b^2, b^4, b^8, b^{16}...\}$

Repeated Squaring (cont'd)

- How to get $\{b^1, b^2, b^4, b^8, b^{16} \ldots\}$
  - — Repeated squaring!
- Then, once you have this, greedily figure out how to compute n
  - — Ex: want $\log_3 4321211$
  - — First compute $\{3^1 = 3, 3^2 = 9, 3^4 = 81, 3^8 = 6561, 3^{16} = 43046721\}$
  - — Then multiply values greedily as long as you don't exceed 4321211

Repeated Squaring (cont'd)

- How to get $\{b^1, b^2, b^4, b^8, b^{16} ...\}$
  - — Repeated squaring!
- Then, once you have this, greedily figure out how to compute *n*
  - — Ex: want $\log_3 4321211$
  - — First compute $\{3^1 = 3, 3^2 = 9, 3^4 = 81, 3^8 = 6561, 3^{16} = 43046721\}$
  - — Then multiply values greedily as long as you don't exceed 4321211
  - — Try 43046721... total would be 43046721... too big
  - — Try 6561... OK! total is 6561
  - — Try 81... OK! total is 6561 * 81 = 531441
  - — Try 9... total would be 531441 * 9 = 4782969... too big
  - — Try 3... OK! total is 531441 * 3 = 1594323
  - — So then $\log_3 4321211$ is 8 + 4 + 1 + 1 = 14 (need an extra 1 since we want ceiling)

Implementing This…

```
public int logBaseB (int b, int n) {
    // vals[i] will store b^(2^i)
    int [] vals = new int[30];
    // pows[i] will store 2^i
    int [] pows = new int[30];

    // repeatedly square accum until we exceed n; in this
    // way, accum is always b^(2^i), curPow is always 2^i
    int i = 0, curPow = 1, accum = b;
    while (true) {
        vals[i] = accum;
        pows[i] = curPow;
        if (accum > n)
            break;
        accum *= accum;
        curPow *= 2;
        i++;
    }
```

Implementing This…

```
public int logBaseB (int b, int n) {
   // vals[i] will store b^(2^i)
   int [] vals = new int[30];
   // pows[i] will store 2^i
   int [] pows = new int[30];

   // repeatedly square accum until we exceed n; in this
   // way, accum is always b^(2^i), curPow is always 2^i
   int i = 0, curPow = 1, accum = b;
   while (true) {
      vals[i] = accum;
      pows[i] = curPow;
      if (accum > n)
         break;
      accum *= accum;
      curPow *= 2;
      i++;
   }
```

Trick so loop never exits here

Tells Java to exit the loop

Question: why do we design the loop in this way?

Implementing This…

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    do {
        if (totSoFar * vals[i] < n) {
            totSoFar *= vals[i];
            powSoFar += pows[i];
        }
        i--;
    } while (i >= 0);

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

This algorithm takes on the order of log (log n) steps! Fast!

Implementing This…

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    do {
        if (totSoFar * vals[i] < n) {
            totSoFar *= vals[i];
            powSoFar += pows[i];
        }
        i--;
    } while (i >= 0);

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

This is a "do-while" loop; stoppage check is executed *after* the body of the loop

What would this look like with a *for* loop instead?

Many Ways To Write Same Loop!

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    for (; i >= 0; i--) {
        // don’t use this factor if we’d meet/exceed n
        if (totSoFar * vals[i] >= n)
            continue;
        totSoFar *= vals[i];
        powSoFar += pows[i];
    }

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

Many Ways To Write Same Loop!

```
public int logBaseB (int b, int n) {
    ...
    // now take the powers we have computed and construct n;
    // it will always hold that totSoFar = b^powSoFar
    int totSoFar = 1, powSoFar = 0;
    for (; i >= 0; i--) {
        // don't use this factor if we'd meet/exceed n
        if (totSoFar * vals[i] >= n)
            continue;
        totSoFar *= vals[i];
        powSoFar += pows[i];
    }

    // we now have the largest power of b that does not
    // reach n, so add one to it and return
    return powSoFar + 1;
}
```

Note use of blank initialization statement

"continue" means to skip rest of loop body

OK, What Have We Missed?

- “Switch”
  - — Generally pretty useless, in my opinion, so won’t talk about it much
  - — Look it up in the book
  - — In a nutshell: alternative to specific form of “if”:

```
if (a == 1) {
   // do something
} else if (a == 2) {
   // do something else
} else if (a == 3) {
   // do another thing
} else {
   // do this
}
```

  - — Switch is a bit more efficient, but easier to screw up
  - — I try to avoid it...

About Does It for Control Flow, Basic Syntax

- Questions?

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Start with array containing 2 through *n*
  - — [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]

A Quick Intro to A0

- You’ll be implementing prime factorization
- Need to write two separate methods
  - — First uses “Sieve of Eratosthenes” to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does “Sieve of Eratosthenes” work?
  - — Have a cursor; number under cursor (and everything to left) is prime
  - — [**2**, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Pass thru array; kill everything (by sliding?) that is a power of num under cursor
  - — [**2**, 3, ~~4~~, 5, ~~6~~, 7, ~~8~~, 9, ~~10~~, 11, ~~12~~, 13, ~~14~~, 15, ~~16~~, 17, ~~18~~, 19, ~~20~~, 21, ~~22~~, 23, ~~24~~]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Pass thru array; kill everything (by sliding?) that is a power of num under cursor
  - — [**2**, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, -, -, -, -, -, -, -, -, -, -, -]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Then move the cursor along
  - — [2, **3**, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, -, -, -, -, -, -, -, -, -, -, -]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Kill everything to right of cursor that's a multiple
  - — [2, **3**, 5, 7, ~~9~~, 11, 13, ~~15~~, 17, 19, ~~21~~, 23, -, -, -, -, -, -, -, -, -, -, -]

A Quick Intro to A0

- You’ll be implementing prime factorization
- Need to write two separate methods
  - — First uses “Sieve of Eratosthenes” to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does “Sieve of Eratosthenes” work?
  - — Kill everything to right of cursor that’s a multiple
  - — [2, **3**, 5, 7, 11, 13, 17, 19, 23, -, -, -, -, -, -, -, -, -, -, -, -, -, -]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < *n*
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Onve number under cursor gets to sqrt(*n*) you are done!
  - — [2, 3, **5**, 7, 11, 13, 17, 19, 23, -, -, -, -, -, -, -, -, -, -, -, -, -, -]

A Quick Intro to A0

- You'll be implementing prime factorization
- Need to write two separate methods
  - — First uses "Sieve of Eratosthenes" to compute an array containing all primes < n
  - — Second uses the array of primes to do prime factorization of a number
- How does "Sieve of Eratosthenes" work?
  - — Once number under cursor gets to sqrt(n) you are done!
  - — [2, 3, **5**, 7, 11, 13, 17, 19, 23, -, -, -, -, -, -, -, -, -, -, -, -, -, -]
- Second thing: use this array to "pretty print" prime factorization

A Primer on Printing

- Java has something called a "PrintStream" that allows character output
- System.out, System.err are both objects of type PrintStream

A Primer on Printing

- Two major PrintStream methods

```
// Prints “This is my string.” followed by return
System.out.println (“This is my string.”);

// Prints “This is my string.” with no return
System.out.format (“This is my string.”);
// Prints “This is my string.” followed by return
System.out.format (“This is my string.\n”);
// Prints “This is my num: 27.” with no return
System.out.format (“This is my num: %d.”, 27);
// Prints “This is my num: 27.” with no return
int i = 27;
System.out.format (“This is my num: %d.”, i);
// Prints “My nums are 27 and 29.” with a return
int i = 27, j = 29;
System.out.format (“My nums are %d and %d.\n”, i, j);
```

Have Fun!