Basic Java Syntax (1)

(and some algorithms as well)

Prof. Chris Jermaine
cmj4@cs.rice.edu

Basic Java Syntax and Control Flow

- 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 observations:

Edit Distance

- Classic algorithm (SW) based on following observations:
  - — ED(string1 + c, string2 + c) = ED (string1, string2)

    Ex: ED(“Christoph”, “Chrisph”) = ED(“Christop”, “Chrisp”)... Why?

Edit Distance

- Classic algorithm (SW) based on following observations:
  - — ED(string1 + c, string2 + c) = ED (string1, string2)
  - — ED(string1 + c, string2) <= ED (string1, string2) + 1

    Ex: ED(“Chrisp”, “Chris”) <= ED (“Chris”, “Chris”) + 1 [can always delete!]

Edit Distance

- Classic algorithm (SW) based on following observations:
  - 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!]

Edit Distance

- Classic algorithm (SW) based on following observations:
  - — 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!]

    Ex: ED(“Chrisa”, “Chrisb”) <= ED(“Chris”, Chris”) + 1

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!]
- How to actually implement this?
- 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 this line?
  - — *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 7)" 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
  - — 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] < 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; many would never do this)

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!