CSE 142

More About Iteration

Introduction

- Review
  - Basic collections - ArrayList
  - Iteration over collections – iterators and while loops
- Today
  - More general patterns of iteration
  - For loops
  - Nested loops
- Reading
  - Dugan notes: ch. 15, 17

Iterating using Iterators

- To process all the elements of an ArrayList (or of any of the other kinds of collections in Java), we can use iterators.
- An ArrayList iterator allow us to
  - go through *all* the elements,
  - *in order* from the first to the last.
- What if we don't want to go through all the elements?
- What if we want to go through them in a different order?

Iterating using Indices

- We can iterate through ArrayLists using integer indices and get( ) messages, instead of Iterators.
- Here's a simple example using indices, which mimics what Iterators can do:

```
ArrayList names = ...;
Iterator iter = names.iterator( );
while (iter.hasNext( )) {
    String name = (String) iter.next( );
    System.out.println(name);

}
```

```
ArrayList names = ...;
int index = 0;
while (index < names.size( )) {
    String name = (String) names.get(index);
    System.out.println(name);
    index = index + 1;
}
```

Iterating Backwards

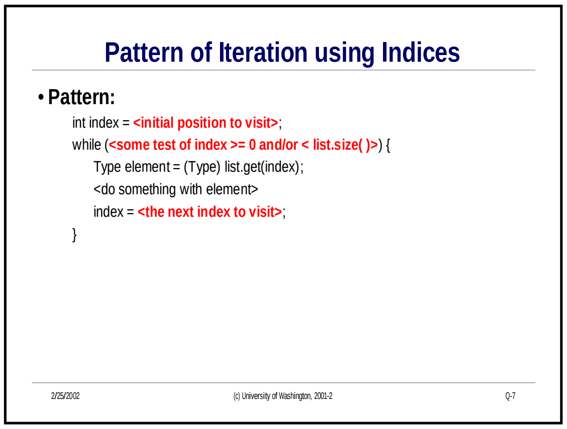
- Here's an example that iterators can't do: iterating in reverse order, last to first:

```
ArrayList names = ...;

int index = names.size( ) - 1;   // start at the last position in the list
while (index >= 0) {             // keep going while we're not before the first element
    String name = (String) names.get(index);
    System.out.println(name);
    index = index - 1;           // visit the previous element next
}
```

Visiting Only Even Elements

- Here's another example that iterators can't do: visiting only the even elements:

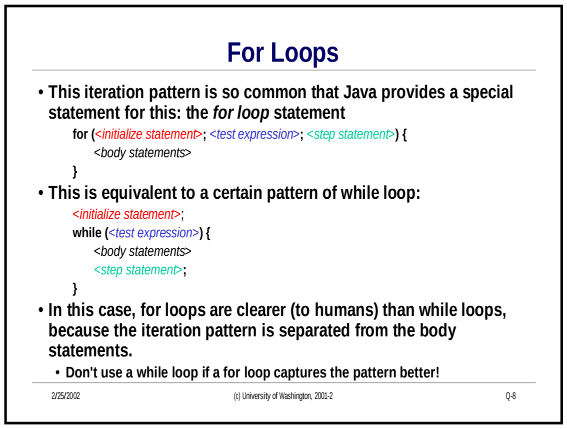
```
ArrayList names = ...;

int index = 0;                        // start at the first even position in the list
while (index < names.size( )) {       // keep going while we're not after the end
    String name = (String) names.get(index);
    System.out.println(name);
    index = index + 2;                // visit the next even element next
}
```



Pattern of Iteration using Indices

- **Pattern:**

```
int index = <initial position to visit>;
while (<some test of index >= 0 and/or < list.size( )>) {
    Type element = (Type) list.get(index);
    <do something with element>
    index = <the next index to visit>;
}
```



For Loops

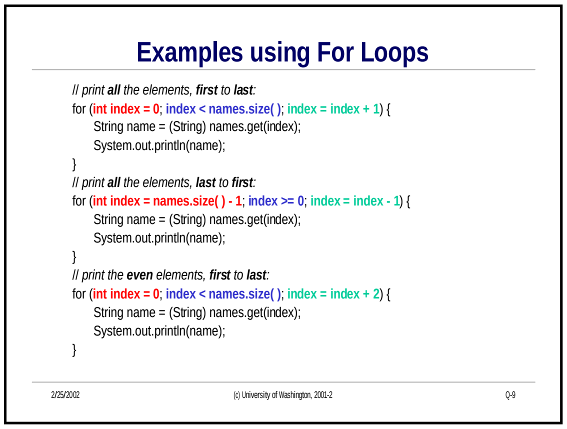
- This iteration pattern is so common that Java provides a special statement for this: the *for loop* statement

```
for (<initialize statement>; <test expression>; <step statement>) {
    <body statements>
}
```

- This is equivalent to a certain pattern of while loop:

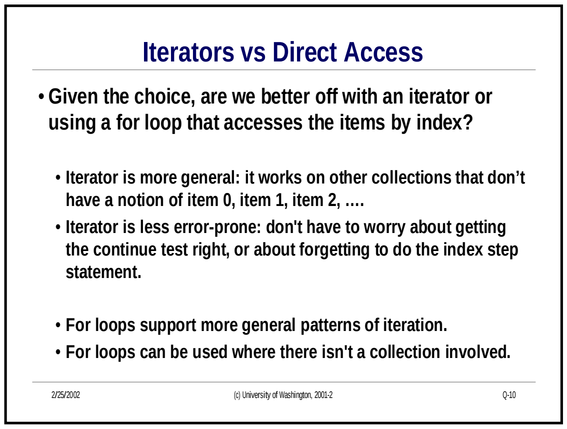
```
<initialize statement>;
while (<test expression>) {
    <body statements>
    <step statement>;
}
```

- In this case, for loops are clearer (to humans) than while loops, because the iteration pattern is separated from the body statements.
  - Don't use a while loop if a for loop captures the pattern better!



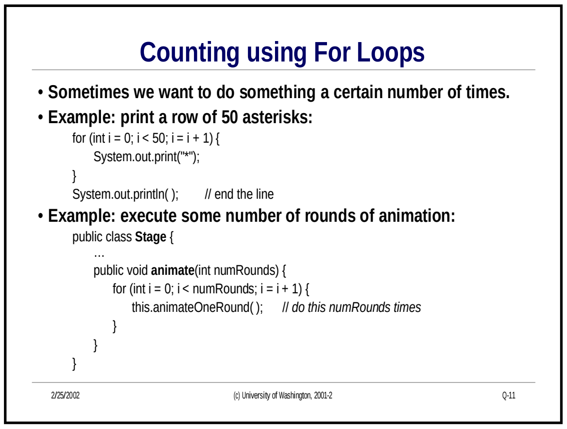
Examples using For Loops

```
// print all the elements, first to last:
for (int index = 0; index < names.size( ); index = index + 1) {
    String name = (String) names.get(index);
    System.out.println(name);
}
// print all the elements, last to first:
for (int index = names.size( ) - 1; index >= 0; index = index - 1) {
    String name = (String) names.get(index);
    System.out.println(name);
}
// print the even elements, first to last:
for (int index = 0; index < names.size( ); index = index + 2) {
    String name = (String) names.get(index);
    System.out.println(name);
}
```



Iterators vs Direct Access

- Given the choice, are we better off with an iterator or using a for loop that accesses the items by index?
  - Iterator is more general: it works on other collections that don't have a notion of item 0, item 1, item 2, ...
  - Iterator is less error-prone: don't have to worry about getting the continue test right, or about forgetting to do the index step statement.
  - For loops support more general patterns of iteration.
  - For loops can be used where there isn't a collection involved.



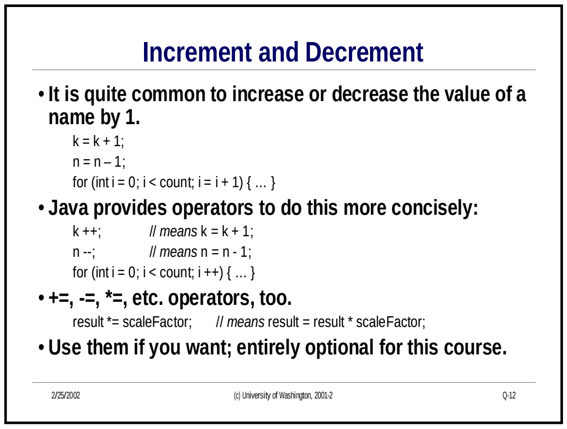
Counting using For Loops

- Sometimes we want to do something a certain number of times.
- Example: print a row of 50 asterisks:

```
for (int i = 0; i < 50; i = i + 1) {
    System.out.print("*");
}
System.out.println( );      // end the line
```

- Example: execute some number of rounds of animation:

```
public class Stage {
    …
    public void animate(int numRounds) {
        for (int i = 0; i < numRounds; i = i + 1) {
            this.animateOneRound( );    // do this numRounds times
        }
    }
}
```



Increment and Decrement

- It is quite common to increase or decrease the value of a name by 1.

```
k = k + 1;
n = n – 1;
for (int i = 0; i < count; i = i + 1) { ... }
```

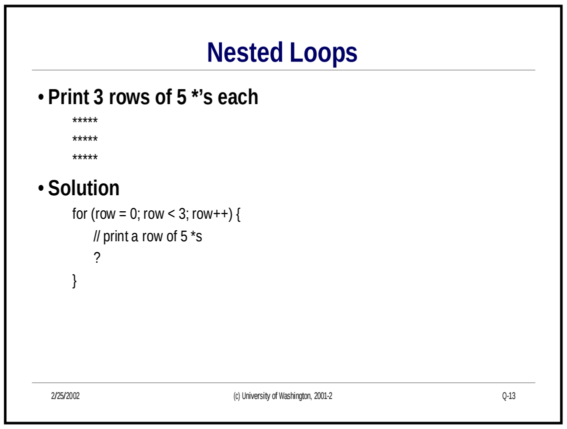
- Java provides operators to do this more concisely:

```
k ++;        // means k = k + 1;
n --;        // means n = n - 1;
for (int i = 0; i < count; i ++) { ... }
```

- +=, -=, *=, etc. operators, too.

```
result *= scaleFactor;     // means result = result * scaleFactor;
```

- Use them if you want; entirely optional for this course.



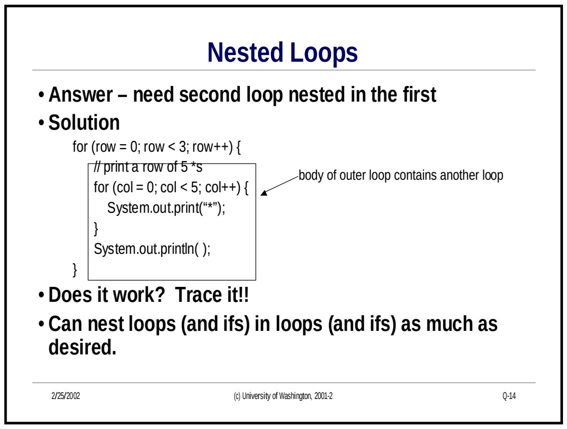
Nested Loops

- Print 3 rows of 5 *'s each

```
*****
*****
*****
```

- Solution

```
for (row = 0; row < 3; row++) {
    // print a row of 5 *s
    ?
}
```



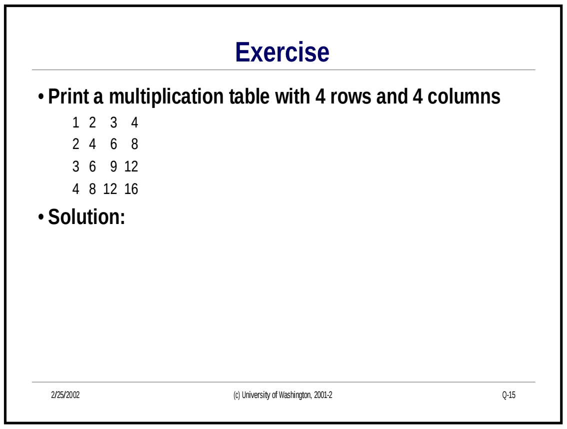
Nested Loops

- Answer – need second loop nested in the first
- Solution

```
for (row = 0; row < 3; row++) {
    // print a row of 5 *s
    for (col = 0; col < 5; col++) {
        System.out.print("*");
    }
    System.out.println( );
}
```

body of outer loop contains another loop

- Does it work? Trace it!!
- Can nest loops (and ifs) in loops (and ifs) as much as desired.



Exercise

- Print a multiplication table with 4 rows and 4 columns

```
1  2   3   4
2  4   6   8
3  6   9  12
4  8  12  16
```

- Solution: