CS415

INTRODUCTION TO COMPUTER SCIENCE

FALL 2017

12 REPETITION AND COLLECTIONS

CHAPTER 11 AND 13

LAST TIME

- Conditions: Boolean expressions
  - relational operators
  - Boolean operators
- Conditionals: modeling decisions
  - if/then
  - if/then/else

PREVIEW

- Programs
  - Control structures
  - Data structures
- Loops
  - definite loops
  - indefinite loops
- Array data structures
- Collection interface

PROGRAMS

- All programs are created from
  - Control structures
    - the order in which statements are executed:
      the flow of control during execution
  - Data structures
    - the organization of multiple data objects into groups or containers (which can be nested)

CONTROL FLOW

- Sequential execution (done this)
  - execute the next statement
- Alternative execution
  - if, if-then-else
  - switch
- Loops (this is new)

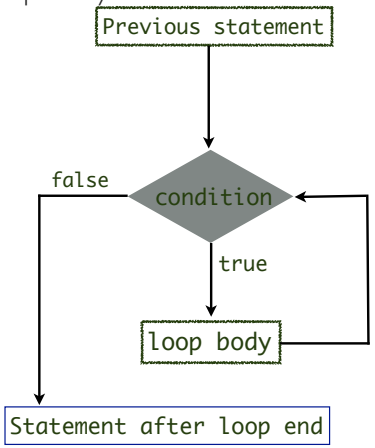
LOOP CONTROL STRUCTURES

- Loops, also called iteration
  - indefinite loop - while and do while
    - don't know number of loop iterations in advance
  - definite loop - for
    - number of loop iterations known before first iteration

WHILE: INDEFINITE LOOP

- while loop executes the loop body as long as the condition remains true; test done at top of loop body

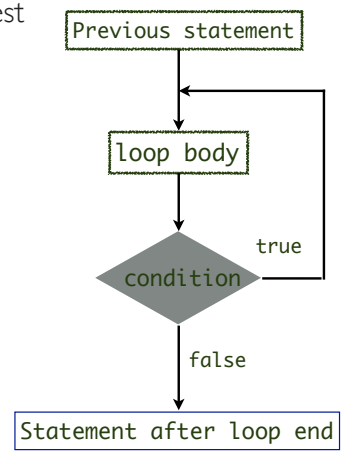
```
previousStatement;
while ( condition )
{
     loopBody;
     //   should change the
     //   condition
}
statementAfterLoop;
```



DO WHILE: INDEFINITE LOOP

- do while loop executes loop body as long as the condition remains true; test done at bottom of loop body; body executed at least once!

```
previousStatement;
do
{
     // loop body
}
while ( condition );
statementAfterLoop;
```



WHILE LOOP EXAMPLE

- While condition is true, keep executing loop "body"

Loop condition

```
outs = 0;
while ( outs < 3 )  // test at "top" of loop
{                   // loop body is between the braces
   batterUp( );
   result = atBat( );
   if ( result >= 1 && result <= 4 )     // a hit
       advanceRunners( result );
   else                                  // an out
       out++;
}
```

Change condition so loop might someday end

FOR LOOP

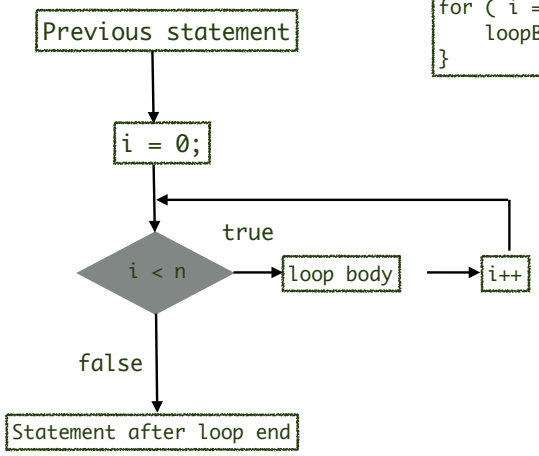
- Execute loop body for fixed number of iterations

```
for ( initialize;  loopCondition;  conditionUpdate )
```

  - initialize is executed once when control flow first reaches the for statement
  - loopCondition must be true for loop body to be executed; it is tested at the "top" of the loop.
  - conditionUpdate is executed at the end of the loop body and typically increments a loop counter variable (i++).

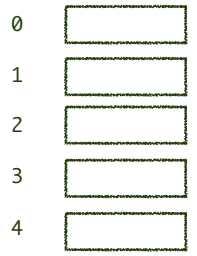
FOR FLOW CHART

```
for ( i = 0;  i < n;  i++ ) {
    loopBody;
}
```



BASIC JAVA ARRAYS

- Array is simplest data structure
- A collection of n objects of the same type
- Declaring an array of *ints* named *arr* with 5 elements:



Both forms are correct

```
int[ ] arr = new int[ 5 ];
```

```
int arr[ ] = new int[ 5 ];
```

- *arr* is a collection of 5 *int* variables

```
int[ ] arr = new int[ 5 ];
```

BASIC JAVA ARRAYS

- Access the individual variables of arr with subscripts

```
arr[ 0 ] = 7;
arr[ 3 ] = 9;
```

0	7
1	
2	
3	9
4	

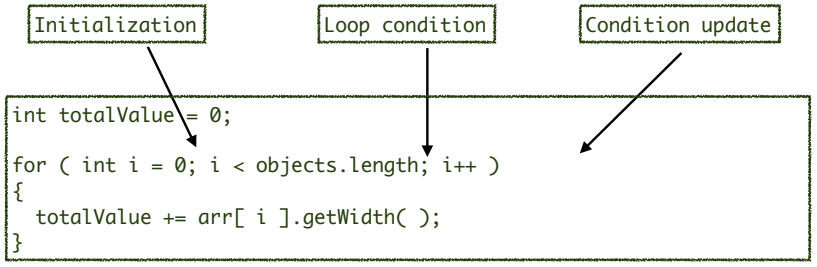
- An array has a public constant variable "length" which is equal to the number of elements in the array.

```
System.out.println( arr.length );
```

5

TYPICAL FOR EXAMPLE

- Do something with each entry in an array
- Compute total width of all Ellipses in array named "arr". (Ellipse arr[ ];)
- Note: the loop variable i is only defined inside the loop; its value cannot be accessed outside the loop.



INSERTION SORT

```
int [] arr = new int[10];
. . .
// insert integer in order in names
void addNumber( int n )
{
   int i = 0;
   while ( i < arr.length &&  arr[i] < n )
      i++;
   // insert at position i, move entries down
   names.add( i, name );
}
```


INSERTION SORT WITH FOR

- Can also use a for in previous example
  - Need a more complex conditionUpdate clause

```
// insert name in alpha order in names
void addName( String name )
{
  int i;
  for( i = 0;
     i < names.size( )
         && name.compareTo( names.get( i )) < 0 );
     i++ )
  ; // there is no loop body!

  // insert at position i, move entries down
  names.add( i, name );
}
```

JAVA ARRAY SUPPORT

- Basic Java arrays have a fixed length; they are difficult to use if you don't know that length when you need to create the array.
- Java's Vector and ArrayList classes create dynamic length arrays: they can grow and shrink as needed during execution.

VECTOR AND ARRAYLIST

- Vector was part of the first release of Java; ArrayList appeared in version 1.2 as part of the Collection framework of classes.
- The Collection interface provides a common interface to a variety of data structures that implement a collection of objects.
- Vector has been retrofitted to implement Collection.

COLLECTION INTERFACE

- Useful methods of the Collection interface for a collection of objects of type T:

```
boolean  add( <T> o );       // add the object to the collection
int      size();             // # elements in collection
boolean  isEmpty();          // true if collection has 0 elements
<T>      get( int i )        // get reference to object i in collection
                             // null returned if there is no object i
void     clear();            // remove all elements from collection
boolean  remove( Object o ); // remove o from collection

Iterator<T> = iterator();    // get an iterator for this collection
```


USING COLLECTIONS

- Defining a Collection variable
  - ArrayList <Player> lineUp;
    - declares a variable named lineUp that can reference an ArrayList object that contains Player objects
  - Vector <Player> lineUp;
    - declares a variable named lineUp that can reference an Vector object that contains Player objects

USING COLLECTIONS

- Creating a Collection object
  - lineUp = new ArrayList <Player>( );
    - creates an ArrayList object with 0 elements
  - lineUp = new Vector <Player>( );
    - creates an Vector object with 0 elements
- If p and q are Player variables:
  - lineUp.add(p); // add object referenced by p to lineUp
    q = lineUp.get(3); // q references element 3 (if it exists)

FOR VARIATIONS

- It is not necessary to process every entry
  - for (int i = 0; i < n; i +=2)
    // do only even entries
- Can also go in reverse order
  - for (int i = n - 1; i >= 0; i--)
    // reverse order: n-1, n-2, ..., 0

RUNNING JAVA FROM THE COMMAND LINE

- Rather than run a java program by clicking on the "run" button in DrJava we can run it from the shell (or interactions pane) command line:

```
> java Program8
```

- Extra information can be added to the command line:

```
> java Program8  hi  7  12.3  xxx
```

- These are called command line arguments.

COMMAND LINE ARGUMENTS

```
public static void main( String[ ] args )
{
....
```

```
> java Program8  hi  7  12.3  xxx
```

- The main method has an array of Strings as a parameter.
  - args[0] references the String "hi"
  - args[1] references the String "7"
  - args[2] references the String "12.3"
  - args[3] references the String "xxx"

get the length of array

ARRAY EXAMPLE

- Convert command line args to an array of ints

```
//    ASSUME: getInt converts a String to an int;
//    if String is not a valid integer,
//    it returns the second (int) parameter  /
//    as a defaultValue.

int[ ] vals = new int [ args.length ];
for ( int i = 0; i < args.length; i++ )
{
   vals[ i ] = getInt( args[ i ], -1 );
}
```


REVERSE FOR

- Process a Vector in reverse order

```
Vector<String> v = new Vector<String>( );

v.add("a ");
v.add("b ");
v.add("c ");

for ( int i = v.size( ) - 1; i >= 0; i-- )
{
  System.out.print( v.get( i ) );
}

Output: c b a
```


MAX VALUE EXAMPLE

```
//Find the maximum width of Ellipses in a Collection
//of Ellipse objects.

  int maxValue = -1;   // widths > 0, so this is safe
  for ( int i = 0; i < objects.size(); i++ )
  {
    int objValue = objects.get( i ).getWidth( );
    if ( objValue >= maxValue )
       maxValue = objValue;
  }
```


MAX VALUE ALTERNATE

```
int maxValue = objects.get( 0 ).getWidth( );

for ( int i = 1; i < objects.size(); i++ )
{
  int objValue = objects.get( i ).getWidth();
  if ( objValue >= maxValue )
      maxValue = objValue;
}
When will this fail?
```

INSERTION

- ArrayList and Vector can insert objects anywhere (part of AbstractList, not Collection)
- add(Object) adds new object at end of array
- To insert name at the start of the names Vector
  - names.add(0, name); // 1st param is position
- Resulting array is the inverse of the order of the add operations. (Can use the reverse for loop to print them out in the original order.)

ITERATORS

- Collection objects can create an associated Iterator object that provides an alternative way to access all objects in the Collection in order

```
Iterator <Player> iter; // iter returns Players
iter  = lineUp.iterator();
while ( iter.hasNext() )
{
  Player p = iter.next(); //get ref to next obj
  p.reportStats();
}
```

get an iterator

any objects left to process?

return next object

ITERATOR TO FIND MAX VALUE

- Use an iterator for finding max value

```
Collection<Treasure> objects = new Vector<Treasure>();
...
   // fill up objects collection
...
Iterator<Treasure> iter;
iter = objects.iterator();

int maxValue = -1;          //values >= 0, so this is safe
while ( iter.hasNext( ) )
{
  Treasure tObj = iter.next( );
  if ( tObj.getValue( ) >= maxValue )
      maxValue = tObj.getValue( );
}
```


MAX VALUE VARIATION

- Suppose want object with max value, not just value

```
Treasure maxObj = null;

Iterator< Treasure > iter = objects.iterator( );

int maxValue = -1;
while ( iter.hasNext() )
{
  Treasure tObj = iter.next();
  if ( tObj.getValue() >= maxValue )
  {
     maxValue = tObj.getValue();
     maxObj = tObj;
  }
}
```

BREAK

- In a switch statement, a break says "break out" of the switch -- start execution at the line following the } for the switch statement.
- break can also be used in loops to terminate execution of the loop (prematurely)

BREAK IN FOR

- Search for key in the Vector, stop if you find it.

```
int i;
for ( i = 0; i < names.size( ); i++ )
{
   if ( key.equals( names.get( i ) )
      break;
}
System.out.println( "Found at " + i );
```

- avoid using break; a compound condition is better

```
int i;
for ( i = 0; i < names.size( )
            && ! key.equals( names.get( i ); i++ )
;
System.out.println( "Found at " + i );
```


CONTINUE IN LOOPS

- continue is similar to break, but it only terminates execution of the current iteration, not entire loop.

```
// don't process names that start with key
for ( int i = 0; i < names.size( ); i++ )
{
   if ( names.get( i ).startsWith( key ) )
      continue;  // skip to next iteration
   // add key to front of name & store it back
   names.set( i, key + names.get( i ) );
}
```


DEBUGGING LOOPS

- Loops increase program complexity significantly.
  - Increase likelihood of bugs
  - Debugging is harder with loops
- Debugging loop techniques
  - Hand simulation - trace execution on paper, recording variable values and changes
  - Debugger tracing - step through executing program with a debugger.
  - Output statements - print key variables used in loop

COMMON LOOP BUGS

- Infinite loop
  - incorrect initialization
  - fail to modify loop condition correctly
  - incorrect method call yields recursion loop
- Loop executes once too many times or once too few. (This is a boundary condition problem.)
  - initial condition or looping condition not correct

REVIEW

- Loops
  - definite loops (for)
  - indefinite loops (while, do-while)
- array data structures
  - basic Java array; args[];
  - Vector and ArrayList classes
- Loop examples

NEXT

- Strings
- Text I/O