CMSC 131: Chapter 21 (Supplement)
Miscellany

Miscellany

Today we discuss a number of unrelated but useful topics that have just not fit into earlier lectures. These include:

**StringBuffer:** A handy class for dealing with strings whose contents can vary dynamically.

**Java’s Stack Data Structure:** One of the simplest data structures in Java’s class library.

**Java’s Method Call Stack:** How does Java keep track of local variables and parameter when methods are called?

### StringBuffer

**The problem with Strings:**

- **Strings are immutable objects.** This means that once constructed it is not possible to change its contents.

- **Example:** Form a string by repeated concatenation:

  ```java
cchar[ ] c = { 'H', 'e', 'l', 'l', 'o' };
  String s = ""
  for ( int i = 0; i < c.length; i++ )
    s += c[i];
```

  This is quite inefficient because it produces one entirely new String object for each new assignment to `s`.

**Q:** What is a more efficient way to do this?

**Ans:** Store `s` as an expandable array of chars. (Double its size whenever we run out of space.) Cast the final array to a String.

**StringBuffer:** Java offers just such a structure for you.
**StringBuffer**

**StringBuffer**: A "mutable" representation of a string, which provides efficient methods for modifying the contents.

**Some StringBuffer Methods**:

- `StringBuffer( )` - Constructs an empty string buffer.
- `append( ... )` - Converts the argument into a string, and appends it to the end of the string buffer. (Possible arguments include boolean, char, double, float, int, long, Object, and char[ ])
- `charAt( int index )` - Returns the character at the specified index.
- `length( )` - Returns the number of characters in the buffer.
- `toString( )` - Returns a String representation of the buffer.

**Example**:

```java
StringBuffer b = new StringBuffer( );
b.append( 99.5 );
b.append( '%' );
b.append( " pure" );
System.out.println( b );
```

**StringBuffer: Example**

**Example**: A method `getWords` which is given a string and strips off spaces and punctuation, converts words to lower-case, and outputs a string with the results.

**Java Class Library Utilities**:

- `String split`: Splits a string about a given regular expression pattern.
  - `[abc]` matches characters 'a', 'b', 'c'.
  - `[abc]*` matches 0 or more repetitions of these characters.
  - `[abc]+` matches 1 or more repetitions of these characters.
  - To remove punctuation we split about 1 or more occurrences of space, comma, period, question mark: `split( "[ ,.?]+" )`
- `String toLowerCase`: Returns an equivalent lower-case string.
- `String valueOf`: Produces a String representation of an object.
public static String getWords( String s )
{
    String[] words = s.split( "\[ ,.\?]+" );
    StringBuffer buffer = new StringBuffer( );
    for ( int i = 0; i < words.length; i++ )
    {
        buffer.append( words[i].toLowerCase() );
        if ( i < words.length-1 ) buffer.append( " ");
    }
    return String.valueOf( buffer );
}

public static void getWordsTest( )
{
    String s1 = "Do you wake up in the morning feeling sleepy and grumpy?";
    System.out.println( "\[" + getWords( s1 ) + "]" );
    String s2 = "Then you must be Snow White.";
    System.out.println( "\[" + getWords( s2 ) + "]" );
}

Stacks

Stack: A stack is an abstract data structure for storing a collection of items. Items can be inserted into the stack and removed from the stack, but the rule is the most recent item inserted is the first item to be removed. (Last in, first out)

Intuition: Think of it like a stack of plates in a restaurant. Items:

- can be inserted (or pushed) onto the top of the stack.
- can be removed (or popped) off of the top of the stack.
- insertions/removals from other positions are not allowed.

Stack Operations

Stack Operations: An abstract (mathematical) stack supports:

- **push(x)**: inserts item x at the top of the stack
- **pop()**: removes the item at the top of the stack (if one exists) and returns its value.
- **top()**: returns the value of the item at the top of the stack, without removing it.
- **empty()**: returns true if the stack is empty

Java's Stack class: (in java.util) Java provides a Stack, with the following corresponding operations.

- **Stack()**: creates an empty stack
- **push(Object x)**: pushes x on the stack
- **pop()**: pops the stack and returns its value. (Exception if empty)
- **peek()**: returns (without removal) the top value of the stack. (Exception if empty)
- **empty()**: returns true if the Stack is empty.
Java Memory Layout

Memory Layout: We mentioned before that Java splits memory into two major sections:

Local storage: This is where your local variables (including parameters) are stored.
Heap: This is where all objects (created by new) are stored.

Q: When one method calls another, how does Java save all the local variables until returning?

Java Call Stack: The local variables for each method are stored on a stack.

- whenever a new method is called, its local variables (including parameters) are pushed onto the stack
- when the method returns, the local variables are popped off the stack (and hence are no longer accessible).

Call Stack: Example

```java
public class CallStack {

    public static int numberLowerCase( String theStr ) {
        int count = 0;
        for ( int i = 0; i < theStr.length(); i++ )
            if ( Character.isLowerCase( theStr.charAt( i ) ) ) count++;
        return count;
    }

    public static void stats( String str ) {
        int total = str.length();
        int lower = numberLowerCase( str );
        System.out.println( "String: " + str);
        System.out.println( "Total count: " + total);
        System.out.println( "Lower case count: " + lower );
    }

    public static void main( String[ ] args ) {
        CallStack.stats( "Que Bueno!" );
    }
}
```
Call Stack: Example

```java
public static int numberLowerCase(String theStr) {
    int count = ... int i = ...
}
public static void stats(String str) {
    int total = ...
    int lower = numberLowerCase(str);
    ...
}
public static void main(String[] args) {
    CallStack.stats("Que Bueno!");
}
```

(Figure omitted)

Operators Revisited

**Operators:** We discussed various operators (+, -, *, <, ==, &,&, ||) earlier this semester. We omitted discussion of a few.

- **Bitwise operators:** Operate on values as binary numbers.
- **Conditional operator:** An "if-then-else" operator.

**Bitwise Operators:** Recall that all quantities are stored as binary numbers in memory. For example:

```java
int x = 1037;  // binary: ...0010000001101 filled out to 32 bits
char c = 'y';  // binary: ...0000001111001 filled out to 16 bits
boolean b = true; // binary: 1
```

You are not required to know how these conversions are performed. (It is covered in later courses.)

Java's bitwise operators act on these binary representations.

**Bitwise Operators**

Java supports the standard bit operators:

- ~a: complement of a
- a & b: and (1 if both a and b are 1)
- a | b: or (1 if either a and b are 1)
- a ^ b: exclusive or (1 if either a or b is 1, but not both)
Bitwise Operators

Java's bitwise operators can be applied
- to any integral type: char, byte, short, int, long
- to Boolean

When applied to integral types, the operations are applied bitwise:

```java
int a = 45; // a = ...00101101
int b = 14; // b = ...00001110
int c = a & b; // c = (00101101 & 00001110) = 00001100 (= 12)
```

Who uses these: They are used in often hardware-related tasks (device management) and have other surprising uses. (E.g.: Using exclusive-or you can swap to integers without a temporary.)

Shift Operators

Another common operation involves shifting bits left or right.

- `a << b`: Shift a left by b positions
- `a >> b`: Shift a right by b positions (filling with the sign bit)
- `a >>> b`: Shift a right by b positions (filling with 0's)

Notes:
- `a` must be integral type (byte, short, ..., long).
- `b` should be a nonnegative integral type.

Sign bit: Because there is no "-" sign in binary, Java encodes negative numbers using a method called 2's-complement representation. We will not discuss this, but a key element is that the leftmost bit, called the sign bit, is:

- 0 for positive numbers
- 1 for negative numbers

We often want to keep the sign bit unchanged when shifting.
Shift Operators

Example: Rather than use 32-bit int's, we use a 10-bit example.

```java
int a = ... // a = 1100101101
int b = 3;
int c = a << b; // c = 0101101000
int d = a >> b; // d = 1111100101
int e = a >>> b; // e = 0001100101
int f = ... // f = 0100101101
int g = f >> b; // g = 0000100101
```

Conditional Operator

Conditional Operator: This is a handy operator, which acts like a little if-then-else statement within an expression.

```
(boolean-condition) ? (true-value) : (false-value)
```

Example: Absolute value method. If the argument x is negative, then return \(-x\), otherwise return x. Without conditional:

```java
public static int absValue1( int x ) {
    if ( x < 0 ) return -x;
    else return x;
}
```

With the conditional operator: you can put the whole expression on one line.

```java
public static int absValue2( int x ) {
    return ( x < 0 ? -x : x);
}
```
Conditional Operator

Example: Set max to the maximum of x and y.

    double max = (x > y) ?  x : y ;

Example: If String s is "zero", then set x to 0.0, and otherwise set it to 3*(z + 13.2):

    String s = ...
    double z = ...
    double x = s.equals("zero") ?  0.0 : 3*(z + 13.2);

ArrayList

The Problem with Arrays:

   Resizing: Arrays are not suitable for situations where the size of the array changes frequently.
   Appending to an Array: if we reach the maximum capacity of an array and we need to add an
   element, we have to create a new array, copy over elements, and add the desire element.

ArrayList:

   - A class in the Java class library that implements a resizable array.
   - It is part of the java.util package, and therefore an appropriate import statement is required.
   - An ArrayList holds generic Object references. Each element is:
     - a reference to any class object or array
     - any primitive type, by first putting it in a wrapper, such as Integer.
   - When an element is removed from the array, you must explicitly cast it back to its original
     type.
ArrayList Methods

Some of ArrayList methods:

- `ArrayList( )`: Initializes an array list of size 0.
- `add( Object obj )`: Adds the object to the end of the array. (Automatically expands the array if needed.)
- `add( int i, Object obj )`: Adds the object at position i (and shifts remaining objects down to make room).

The following return an object of type `Object` or `Object[]`. You must cast to the original type:

- `remove( int i )`: Removes the element at index i. (Shifts the remaining elements to close the gap.)
- `get( int i )`: Returns a reference to the element at index i.
- `toArray( )`: Returns a (standard) array with all the elements.

- `clear( )`: removes all the elements from ArrayList.
- `size( )`: returns the number of elements in ArrayList.

ArrayList Example

Here is an example using an `ArrayList` of `Strings`:

```java
ArrayList a = new ArrayList( );
a.add( new String( "Bob" ) ); // [Bob]
a.add( new String( "Carol" ) ); // [Bob, Carol]
a.add( 1, new String( "Ted" ) ); // [Bob, Ted, Carol]
System.out.println( a.size( ) ); // prints: 3
String x = a.get( 2 ); // illegal: cannot convert Object to String
String y = (String) a.get( 2 ); // okay: returns "Carol"
a.clear( ); // clear it out
System.out.println( a.size( ) ); // prints: 0
```