CMSC 131: Chapter 16 (Supplement)
Arrays II

Command-Line Arguments

Dissecting main: Recall the main method declaration:

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

- **public**: externally visible.
- **static**: not associated with any one particular object instance.
- **void**: returns no type.
- **String[] args**: is called with an array of String command-line arguments. What are these arguments?

**Command-Line Arguments**: On Unix- and DOS-style systems, when a program is run from the command prompt, options are included on the command line. For example, the Unix command:

    % emacs fooBar.java

runs the **emacs** program on the file **fooBar.java**. The string "**fooBar.java**" is a command-line argument to the program.

Command-Line Arguments

**Command-Line Arguments**: Provide a way for the user running your program to pass in run-time information.

- **run-time options**: affect how the program runs (e.g., run the program in "debug mode" with additional diagnostic output.)
- **I/O file names**: provide the names of files used for input and/or output.
- **special definitions**: define special values (e.g., paper size for a word processor)

**Java Command Arguments**:

- If you run Java directly from the **command prompt**, these arguments are typed right after the name of your Java program:
  
  ```
  javac CommandArgTest.java          (compile your program)
  java CommandArgTest -f foo -b bar  (run it)
  ```

Here "-f", "foo", "-b", "bar" are the (4) command-line arguments.
Command-Line Arguments: Eclipse

Java Command Arguments:
If you run Java from Eclipse, these arguments can be specified when you select "Run..." (rather than "Run As"). [Image omitted from notes]

Command-Line Arguments: Example

```java
public class CommandArgTest {
    public static void main(String[] args) {
        System.out.println("Command line arguments:");
        for (int i = 0; i < args.length; i++)
            System.out.println(" Argument[" + i + "] = " + args[i]);
    }
}
```

Command line arguments:
- Argument[0] = -f
- Argument[1] = foo
- Argument[2] = -b
- Argument[3] = bar

Arrays as Instance Variables

We have discussed storing object references in an array. We can also have an array as an instance variable within a class object.

Example: Email manager. Consists of:

- **Helper class** (EmailMessage) stores:
  - Address field (e.g. "bob@yahoo.com") as a String
  - Message body as a String

- **State** (private data):
  - Array of email messages (msgs) of type EmailMessage, and
  - The number of current messages (nMsgs)

- **Behaviors** (public methods):
  - Constructor (given maximum number of allowed emails)
  - Add an email to the list
  - Delete an email from the list (given its index)
  - Clear the entire list
  - ... (and some others)
Email Manager: General Structure

EmailMessage: A helper class to store email addresses and bodies.

    private String addr: email address
    private String body: email body

MailManager: We utilize a partially filled array. We do not store values in all the entries, only in a specified number:

    private EmailMessage[] msgs: Array of Email messages
    private int nMsgs: Number of active messages

Email Manager: EmailMessage

EmailMessage: Stores a single email message. It provides the following public methods:

    EmailMessage( String a, String b ): Standard constructor is given the address a and body b.
    EmailMessage( EmailMessage e ): Copy constructor is given an email message e.
    String toString( ): Converts to string.
    From: <john@notReal.com> Body: [Hi Everybody]

We omit the implementation details.

MailManager (Part 1)

MailManager: Stores the list of email messages and the current number of active messages. Let us investigate its various methods:

    public MailManager( int max ): Constructor is given the maximum number of emails allowed. It allocates the array for the messages and sets the current number of messages to 0.
        msgs = new EmailMessage[max];
        nMsgs = 0;

    public boolean isFull( ): This tests whether the email array is full.
        return ( nMsgs >= msgs.length );

    public boolean addMsg( EmailMessage m ): Adds a given message to the end of the list (nMsgs) and increments the size (nMsgs++). If the list is full, it returns false, and otherwise it returns true.
        if ( isFull() ) return false;
        msgs[nMsgs++] = new EmailMessage( m );
        return true;
MailManager: Class Definition (Part 1)

```java
public class MailManager {
    private EmailMessage[] msgs; // the messages
    private int nMsgs; // current number of messages

    public MailManager( int max ) {
        msgs = new EmailMessage[max];
        nMsgs = 0;
    }

    public boolean isFull() { return ( nMsgs >= msgs.length ); }

    public boolean addMsg( EmailMessage m ) {
        if ( isFull() ) return false;
        msgs[nMsgs++] = new EmailMessage( m );
        return true;
    }

    // ... more to come ...
}
```

MailManager (Part 2)

**MailManager**: Next we consider the removal methods for deleting a single message and clearing the whole list.

```java
public boolean deleteMsg( int d ) : This deletes a single email at index d. If d is out of range, we return false. Otherwise, we eliminate the entry by sliding the subsequent emails down by one.
    for ( int j = d+1; j < nMsgs; j++ )
        msgs[j-1] = msgs[j];
    nMsgs--;

public void clear() : Clears the entire list.
    for ( int i = 0; i < nMsgs; i++ )
        msgs[i] = null;
    nMsgs = 0;
```

Q: Setting nMsgs to 0 would have been sufficient. Why go to the extra effort of setting them to null?
Ans: By unlinking them, we make it possible for the garbage collector to remove them.
public class MailManager {
    private EmailMessage[] msgs; // the messages
    private int nMsgs; // current number of messages

    // ... construct and add omitted ...

    public boolean deleteMsg( int d ) {
        if ( d < 0 || d > msgs.length ) return false;
        for ( int j = d+1; j < nMsgs; j++ )
            msgs[j-1] = msgs[j];
        nMsgs--;
        return true;
    }

    public void clear( ) {
        for ( int i = 0; i < nMsgs; i++ )
            msgs[i] = null;
        nMsgs = 0;
    }
    // ... more to come ...

    MailManager (Part 3)

    MailManager: Finally, consider toString and an accessor, getMessages, which returns an array of email messages.

    public String toString( ) : Converts the messages into a string.

    public EmailMessage[ ] getMessages( ) : Returns an array containing all the email messages.

    EmailMessage[ ] result = new EmailMessage[nMsgs];
    for ( int i = 0; i < nMsgs; i++ ){
        result[i] = new EmailMessage( msgs[i] );
    }
    return result;
MailManager: Class Definition (Part 3)

```java
public class MailManager {
    private EmailMessage[] msgs; // the messages
    private int nMsgs; // current number of messages

    // ... prior methods omitted ...

    public String toString() {
        String result = "Mailbox: ";
        if (nMsgs == 0) return result + "empty";
        for (int i = 0; i < nMsgs; i++)
            result = result + "\n " + msgs[i];
        return result;
    }

    public EmailMessage[] getMessages() {
        EmailMessage[] result = new EmailMessage[nMsgs];
        for (int i = 0; i < nMsgs; i++)
            result[i] = new EmailMessage(msgs[i]);
        return result;
    }
}
```

MailManager: Sample Driver

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

    MailManager myMail = new MailManager( 5 );

    myMail.addMsg( new EmailMessage("john@notReal.com", "Hi Everybody") );
    myMail.addMsg( new EmailMessage("rose@fantasy.com", "I hate spam") );

    System.out.println( myMail );
    System.out.println( myMail.getMessages() );

    myMail.addMsg( new EmailMessage("pete@imaginary.com", "Me too") );
    myMail.deleteMsg( 0 );

    System.out.println( myMail );
    myMail.clear();

    System.out.println( myMail );
    System.out.println( myMail.getMessages() );
}
```
Deep/Shallow Copying and Privacy Leaks

Deep copying: Make a copy of all objects. This is what we implemented. This is always safe because changes to the copied data cannot affect the original object's integrity.

```java
EmailMessage[ ] myMessages = myMail.getMessage();

EmailMessage[ ] result = new EmailMessage[nMsgs];
for ( int i = 0; i < nMsgs; i++ )
    result[i] = new EmailMessage( msgs[i] );
```

Deep/Shallow Copying and Privacy Leaks

Very shallow copy: What if we just copy the array reference?

```java
result = msgs;  // Shallow copy! Dangerous
return result;
```

There are two problems with this:

- Minor problem: Gives the entire array, not just the current messages.
- Major problem: We give the class user a pointer directly to our private data (msgs). Now they can do whatever they want to it! This is called a privacy leak: private data is directly accessible to the outside world.

Deep/Shallow Copying and Privacy Leaks

Half-deep copy: Copy the array, but do not copy the underlying objects.

```java
EmailMessage[ ] result = new EmailMessage[nMsgs];
for ( int i = 0; i < nMsgs; i++ )
    result[i] = msgs[i];  // Shallow: Copies a pointer to msgs[i]
```

Not always bad: It is harmless if it is not possible to change the underlying objects (e.g., String). Otherwise it is harmful.

Classes that cannot be changed after creation are immutable.