Albert Wallace
January 30, 2012
Fundamentals of Computing II

Text Overview of Plans for Assignment #1
(content roughly equivalent to PowerPoint presentation)

Presented with an opportunity to implement an encryption algorithm, this bit of text will begin to examine one possible design to accomplish such a task. There are no diagrams, there is no sample code (mainly because it is very choppy, and these notes are also full of notes to myself on what I must do to fill in all the holes/chasms/voids in my existing code), and there is not much flash or fluff.

This approach feels modular, harebrained, and perfect for the amount of mystery surrounding this method of encryption. It is a bit messy, what with methods being called back and forth, but by compartmentalizing methods, finding an error becomes easier and it's easy to understand on a human level on a step-by-step basis.

For example, the work done with the deck is contained mostly within the Deck class, with only the methods being called by outside methods/classes; this leaves the encryption/decryption to the WSS class. Similarly, the creation of a card within a deck is left to the Card class, which avoids the need to hardcode 54 different cards into a piece of code on a given storage medium.

As I hinted at, there will be 3 classes: a Card (or Cards) class, a Deck class, and the requisite WSS class. 54 cards form a deck, and the deck is used for processing within the WSS class. For further explanation:

Cards class:

When instantiated, call a constructor to instantiate a single card.

Each Card object will hold the value of the card relative to the suite, the value of the card relative to the WSS algorithm, the "friendly name" of the card (e.g. Jack, Queen, etc.), and the friendly name of the suite (Diamonds, Spades, etc.), and color of the suite. This is all so the deck could be searched by humans, with a little building upon the code. (Also, it's so I can actually see what card is being put where as I code and use the debugger. It provides tangibility, for a little extra effort).

The ideal instantiation would have us get the card number and the suite type, from which we could determine the true card value (relative to the WSS algorithm), friendly name of the card, and color (red or black). Again, a large chunk of this is for my benefit, to make the program feel more tangible.

Deck class
--deckOut method:

Takes a pre-defined array list, ideally with 54 slots [indices 0 through 53]. These slots will represent all the cards, as if they were a real deck. (As created when the constructor is called, the list will be empty & will be a flexible ArrayList, so the number of slots are not hard-coded. This aids in modification operations, as ArrayList objects prove easier to manipulate given built-in Java methods).
The initial array will not be hardcoded, but will be filled on-the-fly in memory as coded in the method.
(To fill the slots, repeat 1 through 13 as the number in.
..Then you can repeat the suite type until you reset the number
..And then you can fill the array. (Or, rather, the array will have been filled)).
—This method may be called upon by the assigned "initDeck" method but does NOT replace it; initDeck will still be used in "WSS" to maintain compatibility, as well as include any necessary functionality to avoid overwriting the last deck. (The last known deck will be kept in memory each time initDeck is run; this is just going to be to help with debugging & practical understanding).
—This approach would have worked equally as well as hardcoding an initial array, but this ray, 1 less array is necessary; we will be constantly working with only that main array, and need not have another array of 52 Card objects in memory in case the working array must be reset.

--[methods to manipulate the deck]

—To find the jokers OR the keystream card, there will be one method (tentatively called jokerSearch), with the argument being the promary objective (a specific joker, or the first joker/last joker found). It will return an integer, $0 \leq x \leq 53$ (or 1 to 54, representing card position) if the search operation can be carried out, or "-1" if: [*]the keystream card is a joker (as the keystream card cannot be a joker). [*] the position of a given card cannot be found (invalid Card array, invalid search parameter). Cards will be searched linearly, and will be searched based on “card name” being joker, where applicable.

For the keystream card, the first card will be checked for a rough index of where to find the keystream card, at which point control will be returned to keyValue (below) to actually determine the value.

—To do a triple cut, there will be one method with its own temporary arrays to rearrange the primary array.

—To do a count cut, there will be one method with its own temporary arrays to rearrange the primary array. It will take input from the joker search method and analyze it, rather than relying on jokerSearch to output a "-1" in the event that there is a joker at the bottom of the deck (since it is still a valid search, but no swapping has to be done).

—To retrieve keystream value, there will be one method, keyValue. The return from jokerSearch (ideally the indexed position of the keystream card) will be input, while output will be information retrieved from that "card" index in the array. This method could easily be responsible for processing which card would be the

keystream card, but it's purely personal preference to put the search in the jokerSearch class.

WSS class (as per assignment, plus extra)

—deck of cards. (created before the constructor is even called)

—constructor. Will NOT create the instance of the deck, but WILL call on the deck's constructor to set up the initial deck of 54 cards. (Also: will ensure 54 cards have been created. I need to check myself as much as possible). Will, by extension of the Deck constructor, call on the deckOut method.

—initDeck. This will call on the deck's deckOut method directly, and will ensure the deck created by WSS is initialized properly.

—Encrypt. After getting keystream values from the Deck (via the jokerSearch & keyValue methods) & storing those in an array for the length of the text, it will take the text, feed it to a toNumbers method, retrieve *an array* of numbers, do calculations by sending both arrays to a secretSum method, then will take that array, throw it back into toNumbers, and will retrieve the stream of characters as a stream value. This will be the encrypted information. This method will do the padding (to ensure the multiple of 5 rule for character length of a message)

—toNumbers. Converts a string to numbers for use with encrypt, and (thanks to overloading) will also take an array of numbers to produce a string of text as a return.

—secretSum. Will take the arrays produced by the Encrypt method and produce one final array representing the encrypted code that has yet to be returned to (somewhat) human-readable letters.

It is possible that search operations could be optimized by predicting where certain cards may be for up to a certain number of steps & hardcoding those in; this would be akin to a human shuffling decks so frequently that they would be able to tell you the result before you could do the first cut. It would save time, but is not necessary for this code.

Also missing in this code is the methods required for decryption. Although it would be simple to work backwards, given that you're always starting with the same fresh deck (ideally), it would still be a bit taxing to ensure proper conversion back. No plans have been made either way to provide a decryption tool--especially one intelligent enough to lop off X values (though I'm sure data could be appended as a method of removing padding, restoring proper case, restoring spacing, etc).

However, as proposed, with all these classes, all these methods, and more (yet to be determined) code, it will be possible to take a word or phrase and encrypt it so well that even the Joker himself would want to be proud. Maybe. Or at least the assignment will be complete.