Assignment 1.4: Arrays and Object Oriented Programming

Due: 8am Monday, October 13th, 2008

Directions

The problems for this assignment are described below. You will submit this assignment by emailing it to me as a Zip file. The file will be named **Assignment-1.4-LastName.zip**, where [LastName] is obviously your last name. All of your submitted assignments will be named in this fashion. The Zip file will contain a folder of the same name (so, Assignment-1.4-LastName), which will contain a folder for each of the subsequent problems. Each of these subfolders will have the assignment number and name of its particular problem (for example, **Assignment-1.4-sieveEratos** or **Assignment-1.4-cards**). Each of these problem subfolders will contain all of the necessary files for that particular problem. It's very important that you follow all of these naming conventions exactly as specified, or you could lose credit.

Problems

1) sieveEratos

Overview You will create a program that generates all the prime numbers below a given number using a Sieve of Eratosthenes.

Algorithm The sieve works as follows:

1. Write out all the numbers between 1 and N, the upper bound of the prime numbers.
2. Cross out 1 since it is not a prime number
3. Starting with the next number not crossed out (here, 2) cross out all subsequent multiples of that number (so all other even numbers)
4. Go to the next number not crossed out (now, 3) and cross out all its subsequent mutliples (so all other multiples of 3)
5. Continue crossing out multiples of numbers in this fashion until the number you start with is greater than N/2

This algorithm will leave you with only prime numbers in your list.

Implementation For our implementation of this sieve, we will use a boolean array to represent our numbers, e.g.

```
int MAX_NUM = 100;
boolean[] numbers = new boolean[MAX_NUM];
```

each value of which we will initialize to `true`. You will "cross off" numbers by setting their values to `false`. Remember that array indices start counting at 0, so to cross off the number `N`, you would set

```
numbers[N-1] = false;
```

Your program should prompt the user to enter a maximum number and then display the number of prime numbers found and each prime number on a subsequent line.

Methods You should create and use the following methods to help you write the program:

```
/* obviously you will need your main method, but you should
 * only use it to read in the top number and print out the
 * number of primes and the primes themselves
 */
public static void main(String[] args) {
    ...
}

/*
 * the getPrimes method will hold the bulk of the algorithm;
 * it takes topN, the top number, and returns an array of ints
 * representing the list of prime numbers
 */
public static int[] getPrimes(int topN){
    ...
}

/*
 * the initNumArray method simply takes a boolean array, sets
 * all the elements to true, and returns it
 */
public static boolean[] initNumArray(boolean[] numArray){
    ...
}

/*
 * the getPrimesArray method takes the boolean array of ``crossed
 * out'' numbers and the number of prime numbers creates an int
 * array containing the actual prime numbers
 */
public static int[] getPrimesArray(boolean[] numArray, int numPrimes){
    ...
}
```

```
/*
 * the printIntArray method simply takes an int array and prints
 * each number on its own line
 */
public static void printIntArray(int[] intArray){
    ...
}
```

2) cards

Overview You will implement an enum type, Suit, and two classes, Card and Deck, which will will use in Assignment 1.5 to make a Blackjack game.

Suit enum In order to represent the four suits in cards, you will create an enum type called Suit, which has possible values of Hearts, Diamonds, Spades, and Clubs. You will use this type in the Card class to represent the suit of your card.

Card class The Card class will enable us to make each card its own object. The data we need to represent for each card are the card's number and the card's suit. To represent the number, we'll just use an int variable (ranging from 1 for Ace to 2, 3, 4,...12 for Queen, 13 for King). To represent the suit, we'll use our Suit type. So, the data of our Card class might look something like this:

```
private int myNumber;
private Suit mySuit;
```

What methods will you need? You'll need a public constructor that takes the number and suit of the card as arguments, so perhaps

```
public Card(Suit aSuit, int aNumber){...}
```

as well as a public method for getting the full name of the card (like Ace of Hearts or Nine of Diamonds). This method might look something like this:

```
public String getName(){...}
```

And finally, you'll need a method just for getting the numerical value of the card (1, 2,...12, 13), which might look something like

```
public int getNumber(){...}
```

Deck class The Deck class will allow us to use our Card class as one might in an actual card game. The data we need to represent in our deck is fairly simple, namely, our 52 cards. To represent our cards, we will have an array of Card objects as well as a variable to represent how many cards left we have in our deck, both of which might look something like this:

```
private int deckSize = 52;
private Card[] cards = new Card[deckSize];
```

Notice how we use our variable to keep track of the number of cards in the deck to initialize our cards array. The cards array will always have a fixed number of Card slots (here, 52).

You will need the following methods in your Deck clss. The first method you should always write in any class is the no-args constructor, so for us

```
public Deck(){...}
```

which should actually initialize all 52 cards since the `private Card[] cards = new Card[deckSize];` only creates the *space* for the cards but doesn't actually fill that space with any Card objects. You should also write an overloaded constructor that takes a boolean argument to determine whether the deck should also be shuffled after it's created, perhaps something like

```
public Deck(boolean shuffle)...
```

Note, that in your second constructor that takes the boolean shuffle argument, you can *call* the first, no-args constructor with the command `this();`, which is useful in that you don't have to re-write the card-creation code. It might also be useful to separate the shuffling mechanism from the constructor by creating a separate shuffle method like

```
public void Shuffle(){...}
```

Your shuffle algorithm should involve some sort of random number generation. It would probably be useful for debugging purposes to have a method that simply prints out each card in the deck (to make sure you've created your deck with all the correct cards or shuffled them correctly). This printing method might print the number of cards in the deck and each card on a line, like

```
public void printDeck(){...}
```

Your final (and most difficult probably) method will deal a card from the top of the deck by removing whatever card is in the `cards[0]` slot and returning it. So the method will have a return value of `Card` and might look something like

```
public Card dealNextCard(){...}
```

Once you've dealt your card, however, you should have an empty slot (to "empty" a slot, just set `cards[0] = null;`) at the beginning of your array. You don't want this, so you will have to shift all the other cards in the deck up one slot. This is where the number of cards in the deck counter comes in handy, since after dealing a few cards, you'll have some empty(`null`) slots at the end of your `cards` array and will need to keep track of how many actual cards you have.

Tester To test your two classes, create another class called `Tester` which contains your `public static void main` method and the entry point for your program. In this tester program, you should create a shuffled deck object and then print out its cards. Then, you should "deal" a few cards into specific Card variables, like

```
Card firstCard = myDeck.dealNextCard();
Card secondCard = myDeck.dealNextCard();
```

You should print the value of each card dealt and then print the deck again after the dealing to make sure the appropriate cards have been removed from the deck.

Extra Credit Modify the constructors of the Deck class so that they can take any number of decks (52 cards each).