# Chap13Yourname.java

## Objects of arrays

```java
/*
 * With Deck.java
 */

import java.io.PrintStream;

public class Chap13 {

	public static void main(String[] args) {
		PrintStream out = System.out;
```

# 13.1 The Deck class

```java
out.println("13.1 The Deck class");
//Create the deck class. Then create a standard Deck here.
Deck deck = new Deck();
```

# 13.1 The Deck class

```
/*
 * With Chap 13
 */

import java.util.Random;

public class Deck {
	public Card[] cards;
	private Random random = new Random();
```

# 13.1 The Deck class

```
public Deck() {
    this.cards = new Card[52];
    int index = 0;
    for (int suit = 0; suit <= 3; suit++) {
        for (int rank = 1; rank <= 13; rank++) {
            this.cards[index] = new Card(rank,suit);
            index++;
        }
    }
}

public Deck(int n) {
    this.cards = new Card[n];
}
//The value constructor creates arrays with n Card objects.
```

# 13.1 The Deck class

```
//Print the standard deck.
deck.print();
out.println();
```

# 13.1 The Deck class

```
public void print() {
    for (int i = 0; i < this.cards.length; i++) {
        System.out.println(this.cards[i]);
    }
}
```

# 13.2 Shuffling decks

```
//for each index i {
//        randomInt: choose a random number between i and (length - 1);
//        swapCards: swap the ith card and the randomly selected card.
//}
```

//The process of Writing pseudo code first and then writing methods
//to make it work is called top-down development.

//Write the randomInt(int low, int high) and swapCards(int n, int m)
//methods for Deck.java.

# 13.2 Shuffling decks

```
public int randomInt(int low, int high) {
    return
}

public void swapCards(int m, int n) {



}
```

# 13.2 Shuffling decks

```
public int randomInt(int low, int high) {
    return low + this.random.nextInt(high - low + 1);
}

public void swapCards(int m, int n) {
    Card temp = this.cards[m];
    this.cards[m] = this.cards[n];
    this.cards[n] = temp;
}
```

# 13.2 Shuffling decks

```
//Write the shuffle() method for Deck.java.
deck.shuffle();
deck.print();
out.println();
```

# 13.2 Shuffling decks

```
public void shuffle() {



}
```

# 13.2 Shuffling decks

```
public void shuffle() {
    for (int i = 0; i < this.cards.length; i++) {
        swapCards(i, randomInt(i, this.cards.length - 1));
    }
}
```

# 13.3 Selection sort

```
out.println("13.3 Selection sort");
//Put the messed-up deck back in order. We can use the selection sort
//algorithm.

//public void selectionSort() {
//          for each index i {
//                    lowestIndex: Find the lowest card at or to the right of i
//                    swapCards: Swap the ith card and the lowest card found
//          }
//}
deck.selectionSort();
deck.print();
out.println();
```

# 13.3 Selection sort

```
public void selectionSort() {
    for (int i = 0; i < this.cards.length; i++) {
        swapCards(i, indexLowest(i));
    }
}
```

# 13.3 Selection sort

```
public int indexLowest(int i) {




}
```

# 13.3 Selection sort

```
public int indexLowest(int i) {
    int indexLowest = i;
    for (int j = i + 1; j < this.cards.length; j++) {
        if (this.cards[indexLowest].compareTo(this.cards[j]) > 0) {
            indexLowest = j;
        }
    }
    return indexLowest;
}
```

# 13.4 Merge sort

```
out.println("13.4 - 13.6 Merge sort");
deck.shuffle();
deck.print();
out.println();
//We can also use the merge sort algorithm:
```

# 13.4 Merge sort

```
//public void mergeSort() {
//        If the deck has 0 or 1 cards, return itself.
//        subdeck(): split the deck into two, d1 and d2.
//        mergeSort(): sort the subdecks d1 and d2
//        merge(Deck d1, Deck d2): compare the first card from each sorted
//                subdeck and choose the lower one. Add it to the merged deck.
//                Repeat until one of the subdecks is empty. Then take the
//                remaining cards and add them to the merged deck.
```

# 13.4 Merge sort

//First, write subdeck(int low, int high) in Deck.java that returns a new
//deck that contains the specified subset of the deck.

```
public Deck subdeck(int low, int high) {
    Deck sub = new Deck(high - low + 1);


    return sub;
}
```

# 13.4 Merge sort

//First, write subdeck(int low, int high) in Deck.java that returns a new
//deck that contains the specified subset of the deck.

```
public Deck subdeck(int low, int high) {
    Deck sub = new Deck(high - low + 1);
    for (int i = 0; i < sub.cards.length; i++)
        sub.cards[i] = this.cards[low + i];
    return sub;
}
```

# 13.4 Merge sort

```
//		public Deck merge(Deck d1, Deck d2) {
//				//use the index i to keep track of where we are at in the first deck,
//				//and the index j for the second deck
//				int i = 0;
//				int j = 0;
//				//the index k traverses the resulting this deck
```

# 13.4 Merge sort

```
//                      for (int k = 0; k < this.card.length; k++) {
//                              //Compare the two cards d1.cards[i] and d2.cards[j], add the smaller
//                              //          to this deck at position k
//                              //increment either i or j
//                              //If all cards of any subdeck have been added to this deck, add
//                              //          all the remaining cards of the other deck to this deck.
//                      }
//                      //return this deck
//              }
```

# 13.4 Merge sort

```
public Deck merge(Deck d1, Deck d2) {

}
```

# 13.4 Merge sort

```
public Deck merge(Deck d1, Deck d2) {
    int i = 0;
    int j = 0;
    for (int k = 0; k < this.cards.length; k++) {
        if (i == d1.cards.length) {
            if (j < d2.cards.length) {
                this.cards[k] = d2.cards[j];
                j++;
            }
        } else {
```

# 13.4 Merge sort

```
            } else {
                if (j == d2.cards.length) {
                    this.cards[k] = d1.cards[i];
                    i++;
                } else {
                    if (d1.cards[i].compareTo(d2.cards[j]) <= 0) {
                        this.cards[k] = d1.cards[i];
                        i++;
                    } else {
                        this.cards[k] = d2.cards[j];
                        j++;
                    }
                }
            }
        }
        return this;
    }
```

# 13.4 Merge sort

```
public Deck mergeSort() {



}
```

# 13.4 Merge sort

```
public Deck mergeSort() {
    if (this.cards.length <= 1) {
        return this;
    }
    Deck d1 = subdeck(0, (this.cards.length - 1) / 2);
    Deck d2 = subdeck((this.cards.length + 1) / 2, this.cards.length - 1);
    return merge(d1.mergeSort(), d2.mergeSort());
}
```

# 13.4 Merge sort

```
deck.mergeSort().print();
out.println();
```

# Asymptotic analysis of algorithms

//The running time of an algorithm on a particular input is the number of
//primitive operations or “steps” executed. The worst-case running time
//gives us an upper bound on the running time. We usually consider one
//algorithm to be more efficient than another if its worst-case running
//time has a lower order of growth. (CLRS 2.2)

# Asymptotic analysis of algorithms

//The selection sort is not very efficient. To sort n items, it has to
//traverse the array at most n-1 times. Each traversal takes an amount of time
//proportional to at most n. The total time is proportional to
//n * (n – 1)
//We say that the selection sort has a worst-case running time of
//\Theta \left( n^{2}\right)
//(prounced "theta of n-squared").

//The merge sort is more efficient. It has a worst-case running time of
//____________________________

# Asymptotic analysis of algorithms

//The selection sort is not very efficient. To sort n items, it has to

//traverse the array at most n-1 times. Each traversal takes an amount of time

//proportional to at most n. The total time is proportional to

//n * (n – 1)

//We say that the selection sort has a worst-case running time of

//\Theta \left( n^{2}\right)

//(prounced "theta of n-squared").

//The merge sort is more efficient. It has a worst-case running time of

//\Theta \left( n\log_{2}n\right)