What we will cover:
Processing 2d Arrays (cont.)
2d Arrays and Methods
Copying 2d Arrays
String Comparison
Command-line Arguments

Finally, we can randomly shuffle all elements within a 2d array like we did with the single-dimensional arrays:

```java
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        int iRand = (int) (Math.random() * matrix.length);
        int jRand = (int) (Math.random() * matrix[i].length);
        // swap values
        int temp = matrix[i][j];
        matrix[i][j] = matrix[iRand][jRand];
        matrix[iRand][jRand] = temp;
    }
}
```

How does this code guarantee that the indexes are actually within the array's range of indices?

2d Arrays and Methods
We use 2d arrays with methods in the exact same manner that we use single-dimensional arrays. Like single-dimensional arrays, we can manipulate the values inside the array, if necessary, and we can create and return newly formed 2d arrays.

```java
public static void initialize(double[][] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            array[i][j] = Math.random() * 100;
        }
    }
}

public static void printArray(double[][] array) {
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            System.out.print(array[i][j] + " ");
        }
        System.out.println();
    }
}
```

Notice that these are just overloaded versions of the methods we created for working with single-dimensional arrays. We can also create versions of these for integer and string arrays. We call these methods the same way we would with any other method:

```java
double[][] data = new double[5][5];
initialize(data);
printArray(data);
```

which outputs:
Class Exercise 42

Copying 2d Arrays
We can create a method to copy 2d arrays that is very similar to the single-dimensional copy from before. Examine this first attempt at a copy method:

```java
public static double[][] copyArray(double[][] array) {
    double[][] retArray = new double[array.length][array[0].length];
    for (int i = 0; i < array.length; i++) {
        for (int j = 0; j < array[i].length; j++) {
            retArray[i][j] = array[i][j];
        }
    }
    return retArray;
}
```

This code works for square and non-square arrays. Does it work for ragged arrays? Why not?

```java
System.out.println();
double[][] data3 = {
    {1, 2, 3, 4},
    {7, 9, 12, 14, 16},
    {1},
    {-17, -16, -13, 2}
};
printArray(data3);
System.out.println();
double[][] data4 = copyArray(data3);
printArray(data4);
```

Outputs
1.00 2.00 3.00 4.00
7.00 9.00 12.00 14.00 16.00
1.00
-17.00 -16.00 -13.00 2.00

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
at Scratch.copyArray(Scratch.java:75)
at Scratch.main(Scratch.java:114)

Why does this error happen? What are we doing wrong? We can correct this by setting the length of the internal array when we process it:

```java
public static double[][] copyArray(double[][] array) {
    double[][] retArray = new double[array.length][];
    for (int i = 0; i < array.length; i++) {
        retArray[i] = new double[array[i].length];
        for (int j = 0; j < array[i].length; j++) {
            retArray[i][j] = array[i][j];
        }
    }
    return retArray;
}
```
By declaring the length only when we know it, we allow ourselves to process any ragged array.

String Comparison
Comparing strings is different from comparing numerical variables. We can compare numerical values by checking for equality (==, !=) or using the comparison operators (>, >=, <, <=). A string variable is represented in memory as a character array; it is not a primitive type. The basic operators for comparison and equality work by comparing actual numeric values—something we cannot do with strings.

When comparing strings, we need to compare the contents stored inside the memory assigned to the variable reference. There are built-in methods in the String class that allow comparison, but they are slightly different from what we are used to seeing. Instead of the typical static method type that we are comfortable with using, the string comparison operators are instance methods. This means that they operate on actual instances of a class, a single named variable of the correct type. Instead of being called as ClassName.staticMethod(), they are called by varName.instanceMethod().

We use the .equals() and .compareTo() instance methods in the String class to compare string variables. These methods must be called similar to the array's length property. To compare two strings:

```java
String s1 = "a";
String s2 = "b";

if (s1.equals(s2)) {
    System.out.println("Equal");
} else {
    System.out.println("Not equal");
}
```

Outputs:
Not equal

The compareTo() method's behavior is described in the Java API:
http://download.oracle.com/javase/6/docs/api/java/lang/String.html#compareTo(java.lang.String)

As outlined, compareTo() lexicographically compares two strings and returns a positive number if the first string is “larger” than the method argument (the second string), a negative number if the first string is “smaller” than the second string, or 0 if the strings are equal. The method compares the calling string's position in relation to the string argument. Let's look at an example:

```java
String s1 = "a";
String s2 = "b";
String s3 = "a";

System.out.println(s1.compareTo(s2));
System.out.println(s2.compareTo(s1));
System.out.println(s1.compareTo(s3));
```
In the first case, s1 is the first string and s2 is the second. The method asks, where is s1 in relation to s2? Since s2 is b, a is one position less than b. In the second case, we compare s2 in relation to s1; b is one position to the right of a. In the third case, we compare a in relation to a; since they are equal, we get zero.

You should play around with this code, specifically comparing lowercase letters to their respective uppercase value. Referring back to ASCII and Unicode, why are these comparisons not equal?

We can use this information to rewrite our search and sort methods to allow them to work on strings. Look at the converted linear search for strings:

```java
/**
 * Linear search of an existing array of strings
 * @param data the string array to be searched
 * @param key the search key
 * @return the index of the key if found, -1 if not
 */
public static int linearSearch(String[] data, String key) {
    for (int i = 0; i < data.length; i++) {
        if (key.equals(data[i])) {
            return i;
        }
    }
    return -1;
}
```

**Class Exercise 43**

**Command-Line Arguments**

As you should realize by now, the main method takes one parameter, a string array called args. These values are loaded from the command-line if the program is called with any parameters.

```java
public class CommandLine {
    public static void main(String[] args) {
        for (int i = 0; i < args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

Eclipse creates a run configuration specific to each program it runs. If you compile and run this code in Eclipse, you can then open it’s run configuration and add arguments. The program must be run at least once before you can access its run configuration. Go to Run ➔ Run Configurations and find the one for your program.
Click on the Arguments tab and enter your command-line arguments:
You can then choose to Run that program:

```java
class CommandLine {
    public static void main(String[] args) {
        if (args.length > 0) {
            int sum = 0;
            for (int i = 0; i < args.length; i++) {
                sum += Integer.parseInt(args[i]);
            }
            System.out.println("Sum: "+sum);
        } else {
            System.out.println("There were no parameters.");
        }
    }
}
```

Remember that args is just an array. You can make sure that there are command-line elements by checking args.length. Using the string comparison methods that we introduced, you can conditionally process your arguments. You can also convert them into numeric values and use them later in the program.

Notice that you could create a program that only asks for user input if command-line arguments were not used.