Lecture 3
Arrays

Arrays

An array is an object that holds a set of values
Each value can be accessed by a numeric index
An array of length N is indexed from 0 to N-1

<table>
<thead>
<tr>
<th>scores</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>92</td>
<td>79</td>
<td>80</td>
<td>85</td>
<td>94</td>
<td>97</td>
<td>80</td>
<td>92</td>
<td>86</td>
<td></td>
</tr>
</tbody>
</table>

The scores array can hold 10 integers, indexed from 0 to 9
The name of the array is an object reference variable

Arrays

Square brackets (the index operator) are used to refer to a specific element in the array

```java
int num = scores[3];
```

An exception is thrown if you attempt to access an array outside of the range 0 to N-1
This is called a bounds error
Each element of scores can be treated as an individual integer

```java
scores[7] = 83;
```

Arrays

The object reference variable is declared without specifying the size of the array

```java
int[] scores;
```

The variable scores can refer to any array of integers
The size of an array is specified when the array object is created

```java
scores = new int[10];
```

As with other objects, those two steps may be combined

```java
int[] scores = new int[10];
```

In Java it is valid to associate the brackets with the array name in a declaration

```java
int myList[];
```

However, this is not a good idea
It is far more readable to associate the brackets with the element type

```java
int[] myList;
```

Together, they define the type of the variable (an array of int)
Arrays

The array's element type is the type of values it stores
An array can be declared to hold any primitive or object type
Only values consistent with the element type can be stored in it

```java
int[] widths = new int[500];
double[] myArray = new double[20];
boolean[] flags = new boolean[80];
String[] names = new String[150];
```

When an array is created it is filled with the default value for the element type

```java
int[] list = new int[15];
for (int i = 0; i < list.length; i++)
    list[i] = 10;
for (int value : list)
    System.out.print(value + " ");
```

Modifying particular elements of that array:

```java
list[3] = 999;
list[0] = list[5];
list[12] = list[13] + list[14];
for (int value : list)
    System.out.print(value + " ");
```

Arrays

The size of an array is stored in a constant called length
Once an array has been created, its size cannot change
The for and for each loops are often used when processing arrays

```java
int[] list = new int[15];
for (int i = 0; i < list.length; i++)
    list[i] = i + 10;
for (int value : list)
    System.out.print(value + " ");
```

Arrays

An array can also be created using an initialization list, which both
creates the array and fills it with values

```java
int[] primes = {2, 3, 5, 7, 11, 13, 17, 19, 23,
                29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71};
```

If an initialization list is used, the new operator is not
The array size is determined by the number of elements in the list

Two-Dimensional Arrays

A basic array is a single-dimensional array
A two-dimensional array is accessed via row and column and
appropriate for tabular data

<table>
<thead>
<tr>
<th>Movie</th>
<th>Rev 1</th>
<th>Rev 2</th>
<th>Rev 3</th>
<th>Rev 4</th>
<th>Rev 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Godzilla</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Guardians of the Galaxy</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>Jersey Boys</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Maleficent</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>The Expendables 3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>The Fault in our Stars</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>X-Men: Days of Future Past</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Two-Dimensional Arrays
Both row and column indexes begin at 0
Use brackets for each index

All elements must have the same type
The type of a 2-D array of integers is int[][]

As with a 1-D array, the size of each dimension is specified when the array object is created

This array has 12 rows (indexed 0 to 11) and 20 columns (indexed 0 to 19)

You can also use an initialization list to create 2-D arrays

This is essentially a list of lists

Nested loops are often used to access elements in a 2-D array

The following code fills a 2-D array with random double values

scores.length is the number of rows
scores[row].length is the number of columns in that row

Processing the movie reviews:

public static void main(String[] args)
{
    int[][] reviews = { {3, 3, 4, 2, 3},
                       {3, 3, 3, 4, 3},
                       {4, 3, 3, 4, 3},
                       {2, 1, 3, 2, 2},
                       {4, 3, 4, 3},
                       {4, 4, 5, 5, 4} };

    int sum;
    double average;
    ...
Two-Dimensional Arrays

The output includes the reviewer average per movie

<table>
<thead>
<tr>
<th>Movie</th>
<th>Rating1</th>
<th>Rating2</th>
<th>Rating3</th>
<th>Rating4</th>
<th>Rating5</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Godzilla</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>3.0</td>
</tr>
<tr>
<td>Guardians of the Galaxy</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4.4</td>
</tr>
<tr>
<td>Jersey Boys</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.2</td>
</tr>
<tr>
<td>Maleficent</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.4</td>
</tr>
<tr>
<td>The Expendables 3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2.0</td>
</tr>
<tr>
<td>The Fault in our Stars</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.6</td>
</tr>
<tr>
<td>X-Men: Days of Future Past</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>3.4</td>
</tr>
</tbody>
</table>

Two-Dimensional Arrays

In Java, a 2-D array is really an array of arrays

If values is a 2-D array with 5 rows and 7 columns, it could be depicted like this:

```
0  1  2  3  4  5  6
|   |   |   |   |   |   |   |
1  2  3  4  5  6  7
|   |   |   |   |   |   |   |
2  3  4  5  6  7  8
|   |   |   |   |   |   |   |
3  4  5  6  7  8  9
|   |   |   |   |   |   |   |
4  5  6  7  8  9 10
```

Two-Dimensional Arrays

Each row in a 2-D array could be instantiated separately

Each row could therefore have a different number of columns, creating a ragged array

```java
int[][] raggedArray = new int[5][];
raggedArray[0] = new int[4];
raggedArray[1] = new int[7];
raggedArray[2] = new int[8];
raggedArray[3] = new int[5];
raggedArray[4] = new int[6];
```

Two-Dimensional Arrays

The concept of a 2-D array can be generalized into a multidimensional array

```java
int[][][] accidents = new int[12][31][24];
```

This array might be used to store the number of traffic accidents in a town, organized by month, day, and hour

EXAMPLE: PRIME SIEVE
Let's look at an example that uses an array to determine a set of prime numbers.

A prime number is an integer greater than 1 that is only divisible by 1 and itself.

One approach is to eliminate numbers that are multiples of prime numbers.

Such an approach is called a sieve because it sifts out unwanted values.

The algorithm we'll examine is called the Sieve of Eratosthenes after the Greek mathematician who developed it.

Suppose we wanted to find all primes less than 30:

First, eliminate multiples of 2 (after 2):

Then eliminate the multiples of 3:

Then eliminate the multiples of 5, and so on.

You can stop once you pass the square root of the highest value you're considering (30).

We'll use an array of booleans to represent eliminated values. The index will correspond to the value being considered.

The first two elements will be ignored.

Initially, all elements are set to true.

When a value is eliminated, the corresponding element in the array is set to false.

When complete, all indexes that still correspond to true values are prime.

```java
import java.util.Scanner;

public class PrimeSieve {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Enter a positive integer: ");
        int n = in.nextInt();
        boolean[] primes = new boolean[n + 1];
        for (int i = 2; i < primes.length; i++) {
            primes[i] = true;
            ...
        }
        ...
        for (int i = 2; i * i <= n; i++) {
            if (primes[i]) {
                for (int j = i * i; j <= n; j += i) {
                    primes[j] = false;
                }
            }
        }
        System.out.println("The prime numbers less than 
        or equal to "+n+" are:");
        for (int i = 2; i < primes.length; i++) {
            if (primes[i]) {
                System.out.print(i + " ");
            }
        }
        System.out.println();
    }
}
```
Example: Prime Sieve

A sample run:

```
Enter a positive integer: 100
The prime numbers less than or equal to 100 are:
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71
73 79 83 89 97
```

Binary Search

A common activity is to search an array for a target element. A linear search starts at the beginning and examines each element. If the values in the array are already sorted, we can use a binary search. Suppose we wanted to find the value 72 in the following list:

```
10 13 21 22 27 30 35 37 41 46 48 53 57 59 60 61 64 69 72 77 85 88 91 95 97
```

We begin by examining the middle element (59):

```
10 13 21 22 27 30 35 37 41 46 48 53 57 59 60 61 64 69 72 77 85 88 91 95 97
```

Since that's not our target, we have to keep looking. If it's in the list, it must be in the second half. With one comparison, we've eliminated half of our data. Now jump to the middle of the remaining viable candidates (79):

```
10 13 21 22 27 30 35 37 41 46 48 53 57 59 60 61 64 69 72 77 85 88 91 95 97
```

We still haven't found the target element, but we've eliminated another quarter of the data (half of the half). This time, the target (if it's there) will be in the left "half":

```
10 13 21 22 27 30 35 37 41 46 48 53 57 59 60 61 64 69 72 77 85 88 91 95 97
```

When there's an even number of elements, we examine the first of the two "middle" elements. Since 64 is not the target, we narrow the scope again and this time find the value we're looking for:

```
10 13 21 22 27 30 35 37 41 46 48 53 57 59 60 61 64 69 72 77 85 88 91 95 97
```

It took 4 comparisons to find our target in a list of 27 elements. Now imagine searching a list of 10,000 elements, with each comparison eliminating half of the viable candidates. The first comparison would eliminate 5,000 values, the second would eliminate another 2,500, etc. Using a binary search, a list of N values can be completely searched in \( \log_2 N + 1 \) comparisons. For example, a list of 1024 values would require at most 11 comparisons.
Binary Search

```java
public static int binarySearch(int[] list, int target) {
    int low = 0;
    int high = list.length - 1;
    int mid;
    while (high >= low) {
        mid = (low + high) / 2;
        if (target < list[mid])
            high = mid - 1;
        else if (target == list[mid])
            return mid; // target was found
        else
            low = mid + 1;
    }
    return -1; // target is not in list
}
```

The variables low and high represent the search range. The loop continues as long as there is data to search. The middle value is identified using integer division:

\[
\text{mid} = \frac{\text{low} + \text{high}}{2};
\]

When the target is found, its index is returned immediately. Otherwise, the search range is narrowed by adjusting the value of either low or high. If the target is never found, a value of -1 is returned.

The java.util.Arrays class contains several methods that perform a binary search. Our example searched an array of integers. A binary search can be performed on an array of objects if the objects implement the Comparable interface.