Chapter 6

Arrays

Introduction to Arrays

• An array is a data structure used to process a collection of data that is all of the same type
  – An array behaves like a numbered list of variables with a uniform naming mechanism
  – It has a part that does not change: the name of the array
  – It has a part that can change: an integer in square brackets
  – For example, given five scores:
    \[score[0], score[1], score[2], score[3], score[4]\]

Creating and Accessing Arrays

• An array that behaves like this collection of variables, all of type \texttt{double}, can be created using one statement as follows:
  \[
  \texttt{double}[] \texttt{score} = \texttt{new} \texttt{double}[5] ;
  \]
• Or using two statements:
  \[
  \texttt{double}[] \texttt{score} ;
  \texttt{score} = \texttt{new} \texttt{double}[5] ;
  \]
  – The first statement declares the variable \texttt{score} to be of the array type \texttt{double}[]
  – The second statement creates an array with five numbered variables of type \texttt{double} and makes the variable \texttt{score} a name for the array

Creating and Accessing Arrays

• The individual variables that together make up the array are called indexed variables
  – They can also be called subscripted variables or elements of the array
  – The number in square brackets is called an index or subscript
  – In Java, indices must be numbered starting with 0, and nothing else
    \[score[0], score[1], score[2], score[3], score[4]\]
Creating and Accessing Arrays

- The number of indexed variables in an array is called the length or size of the array.
- When an array is created, the length of the array is given in square brackets after the array type.
- The indexed variables are then numbered starting with 0, and ending with the integer that is one less than the length of the array.

\[
\text{score}[0], \text{score}[1], \text{score}[2], \text{score}[3], \text{score}[4]
\]

Declaring and Creating an Array

- An array is declared and created in almost the same way that objects are declared and created:
  - `BaseType[] ArrayName = new BaseType[size];`
    - The size may be given as an expression that evaluates to a nonnegative integer, for example, an int variable.
  - `char[] line = new char[80];`
  - `double[] reading = new double[count];`
  - `Person[] specimen = new Person[100];`

Creating and Accessing Arrays

\[
\text{double[]} \text{score} = \text{new double}[5];
\]

- A variable may be used in place of the integer (i.e., in place of the integer 5 above):
  - The value of this variable can then be read from the keyboard.
  - This enables the size of the array to be determined when the program is run:
    \[
    \text{double[]} \text{score} = \text{new double[count]};
    \]

- An array can have indexed variables of any type, including any class type.
- All of the indexed variables in a single array must be of the same type, called the base type of the array.

Referring to Arrays and Array Elements

- Each array element can be used just like any other single variable by referring to it using an indexed expression:
  - `score[0]`
- The array itself (i.e., the entire collection of indexed variables) can be referred to using the array name (without any square brackets): `score`
- An array index can be computed when a program is run:
  - It may be represented by a variable: `score[index]`
  - It may be represented by an expression that evaluates to a suitable integer: `score[next + 1]`
Using the `score` Array in a Program

- The `for` loop is ideally suited for performing array manipulations:
  ```java
  for (index = 0; index < 5; index++)
      System.out.println(score[index] + " differs from max by " + (max-score[index]));
  ```

Three Ways to Use Square Brackets `[]` with an Array Name

- Square brackets can be used to create a type name:
  ```java
  double[] score;
  ```
- Square brackets can be used with an integer value as part of the special syntax Java uses to create a new array:
  ```java
  score = new double[5];
  ```
- Square brackets can be used to name an indexed variable of an array:
  ```java
  max = score[0];
  ```

The `length` Instance Variable

- An array is considered to be an object
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named `length`
  - When an array is created, the instance variable `length` is automatically set equal to its size
  - The value of `length` cannot be changed (other than by creating an entirely new array with `new`)
    ```java
    double[] score = new double[5];
    ```
- Given `score` above, `score.length` has a value of 5

Pitfall: Array Index Out of Bounds

- Array indices always start with 0, and always end with the integer that is one less than the size of the array
  - The most common programming error made when using arrays is attempting to use a nonexistent array index
- When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be `out of bounds`
  - An out of bounds index will cause a program to terminate with a run-time error message
  - Array indices get out of bounds most commonly at the `first` or `last` iteration of a loop that processes the array: Be sure to test for this!
Initializing Arrays

- An array can be initialized when it is declared
  - Values for the indexed variables are enclosed in braces, and separated by commas
  - The array size is automatically set to the number of values in the braces
    ```java
    int[] age = {2, 12, 1};
    ```
  - Given `age` above, `age.length` has a value of 3

Pitfall: An Array of Characters Is Not a String

- An array of characters is conceptually a list of characters, and so is conceptually like a string
- However, an array of characters is not an object of the class `String`
  ```java
  char[] a = {'A', 'B', 'C'};
  String s = a; //Illegal!
  ```
- An array of characters can be converted to an object of type `String`, however
  ```java
  String s = new String(a);
  ```
  - The object `s` will have the same sequence of characters as the entire array `a` ("ABC"), but is an independent copy
- Another `String` constructor uses a subrange of a character array instead
  ```java
  String s2 = new String(a, 0, 2);
  ```
  - Given `a` as before, the new string object is "AB"
Pitfall: An Array of Characters Is Not a String

- An array of characters does have some things in common with `String` objects
  - For example, an array of characters can be output using `println`
    ```java
    System.out.println(a);
    ```
  - Given `a` as before, this would produce the output `ABC`

Arrays and References

- Like class types, a variable of an array type holds a `reference`
  - Arrays are objects
  - A variable of an array type holds the address of where the array object is stored in memory
  - Array types are (usually) considered to be class types

Arrays are Objects

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
  - An array variable names the array as a single item
    ```java
double[] a;
```
  - A `new` expression creates an array object and stores the object in memory
    ```java
    new double[10]
    ```
  - An assignment statement places a reference to the memory address of an array object in the array variable
    ```java
    a = new double[10];
    ```

Arrays Are Objects

- The previous steps can be combined into one statement
  ```java
double[] a = new double[10];
```
- Note that the `new` expression that creates an array invokes a constructor that uses a nonstandard syntax
- Not also that as a result of the assignment statement above, `a` contains a single value: a memory address or `reference`
- Since an array is a reference type, the behavior of arrays with respect to assignment (`=`), equality testing (`==`), and parameter passing are the same as that described for classes
Pitfall: Arrays with a Class Base Type

- The base type of an array can be a class type
  ```java
  Date[] holidayList = new Date[20];
  ```
- The above example creates 20 indexed variables of type `Date`
  - It does not create 20 objects of the class `Date`
  - Each of these indexed variables are automatically initialized to `null`
  - Any attempt to reference any them at this point would result in a "null pointer exception" error message

Array Parameters

- Both array indexed variables and entire arrays can be used as arguments to methods
  - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument
  ```java
  double n = 0.0;
  double[] a = new double[10];//all elements
  //are initialized to 0.0
  int i = 3;
  ```
- Given `myMethod` which takes one argument of type `double`, then all of the following are legal:
  ```java
  myMethod(n);//n evaluates to 0.0
  myMethod(a[3]);//a[3] evaluates to 0.0
  myMethod(a[i]);//i evaluates to 3,
  //a[3] evaluates to 0.0
  ```
Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
  - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only
  - It does not specify the length of the array

\[\text{Array Parameters} \]

- Arrays of double may be defined as follows:
  \[
  \text{double[]} \ a = \text{new double}[10];
  \text{double[]} \ b = \text{new double}[30];
  \]
- Given the arrays above, the method \text{doubleElements} from class \text{SampleClass} can be invoked as follows:
  \[
  \text{SampleClass}.\text{doubleElements}(a);
  \text{SampleClass}.\text{doubleElements}(b);
  \]
  - Note that no square brackets are used when an entire array is given as an argument
  - Note also that a method that specifies an array for a parameter can take an array of any length as an argument

\[\text{Array Parameters} \]

- The following method, \text{doubleElements}, specifies an array of \text{double} as its single argument:

\[
\text{public class SampleClass}
\{
  \text{public static void doubleElements(double[]} a)
  \{
    \text{int i; for (i = 0; i < a.length; i++)}
    \ a[i] = a[i] \times 2;
    \ . . .
    \}
  \}
\]

\[\text{Array Parameters} \]

Pitfall: Use of = and == with Arrays

- Because an array variable contains the memory address of the array it names, the assignment operator (\text{=}) only copies this memory address
  - It does not copy the values of each indexed variable
  - Using the assignment operator will make two array variables be different names for the same array
    \[
    \text{b} = \text{a};
    \]
  - The memory address in \text{a} is now the same as the memory address in \text{b}: They reference the same array
Pitfall: Use of \( = \) and \( == \) with Arrays

- A **for** loop is usually used to make two different arrays have the same values in each indexed position:
  
  ```java
  int i;
  for (i = 0; 
   (i < a.length) && (i < b.length); i++)
    b[i] = a[i];
  ```
  
  - Note that the above code will not make \( b \) an exact copy of \( a \), unless \( a \) and \( b \) have the same length.

---

For the same reason, the equality operator (\( == \)) only tests two arrays to see if they are stored in the same location in the computer’s memory:

- It does not test two arrays to see if they contain the same values:
  
  ```java
  (a == b)
  ```
  
- The result of the above boolean expression will be **true** if \( a \) and \( b \) share the same memory address (and, therefore, reference the same array), and **false** otherwise.

---

In the same way that an **equals** method can be defined for a class, an **equalsArray** method can be defined for a type of array:

- This is how two arrays must be tested to see if they contain the same elements.
- The following method tests two integer arrays to see if they contain the same integer values:

  ```java
  public static boolean equalsArray(int[] a, int[] b) {
    if (a.length != b.length)  return false;
    else {
      int i = 0;
      while (i < a.length) {
        if (a[i] != b[i])
          return false;
        i++;
      }
      return true;
    }
  }
  ```
Arguments for the Method `main`

- The heading for the `main` method of a program has a parameter for an array of `String`
  - It is usually called `args` by convention.
  
  ```java
  public static void main(String[] args)
  ```
  - Note that since `args` is a parameter, it could be replaced by any other non-keyword identifier.

- If a Java program is run without giving an argument to `main`, then a default empty array of strings is automatically provided.

```
java SomeProgram Hi! there
```

- This will set `args[0]` to "Hi", `args[1]` to "!", and `args[2]` to "there"

- It will also set `args.length` to 3

- When `SomeProgram` is run as shown, its output will be:

  `Hi there!`

Arguments for the Method `main`

- Here is a program that expects three string arguments:

  ```java
  public class SomeProgram {
      public static void main(String[] args) {
          System.out.println(args[0] + " " + args[2] + args[1]);
      }
  }
  ```

- Note that if it needed numbers, it would have to convert them from strings first.

Methods That Return an Array

- In Java, a method may also return an array
  - The return type is specified in the same way that an array parameter is specified

  ```java
  public static int[] incrementArray(int[] a, int increment) {
      int[] temp = new int[a.length];
      int i;
      for (i = 0; i < a.length; i++)
          temp[i] = a[i] + increment;
      return temp;
  }
  ```
Partially Filled Arrays

- The exact size needed for an array is not always known when a program is written, or it may vary from one run of the program to another.
- A common way to handle this is to declare the array to be of the largest size that the program could possibly need.
- Care must then be taken to keep track of how much of the array is actually used.
  - An indexed variable that has not been given a meaningful value must never be referenced.

Accessor Methods Need Not Simply Return Instance Variables

- When an instance variable names an array, it is not always necessary to provide an accessor method that returns the contents of the entire array.
- Instead, other accessor methods that return a variety of information about the array and its elements may be sufficient.

Partially Filled Arrays

- A variable can be used to keep track of how many elements are currently stored in an array.
  - For example, given the variable `count`, the elements of the array `someArray` will range from positions `someArray[0]` through `someArray[count - 1]`.
  - Note that the variable `count` will be used to process the partially filled array instead of `someArray.length`.
  - Note also that this variable (`count`) must be an argument to any method that manipulates the partially filled array.

The "for each" Loop

- The standard Java libraries include a number of collection classes.
  - Classes whose objects store a collection of values.
- Ordinary `for` loops cannot cycle through the elements in a collection object.
  - Unlike array elements, collection object elements are not normally associated with indices.
- However, there is a new kind of `for` loop, first available in Java 5.0, called a `for-each loop` or `enhanced for loop`.
- This kind of loop can cycle through each element in a collection even though the elements are not indexed.
The "for each" Loop

- Although an ordinary for loop cannot cycle through the elements of a collection class, an enhanced for loop can cycle through the elements of an array.
- The general syntax for a for-each loop statement used with an array is:
  ```java
  for (ArrayBaseType VariableName : ArrayName)
  Statement
  ```
- The above for-each line should be read as "for each VariableName in ArrayName do the following:"
  - Note that VariableName must be declared within the for-each loop, not before.
  - Note also that a colon (not a semicolon) is used after VariableName.

The "For-Each" Loop

- The for-each loop can make code cleaner and less error prone.
- If the indexed variable in a for loop is used only as a way to cycle through the elements, then it would be preferable to change it to a for-each loop.
  - For example:
    ```java
    for (int i = 0; i < a.length; i++)
    a[i] = 0.0;
    ```
  - Can be changed to:
    ```java
    for (double element : a)
    element = 0.0;
    ```
- Note that the for-each syntax is simpler and quite easy to understand.

Methods with a Variable Number of Parameters

- Starting with Java 5.0, methods can be defined that take any number of arguments.
- Essentially, it is implemented by taking in an array as argument, but the job of placing values in the array is done automatically.
  - The values for the array are given as arguments.
  - Java automatically creates an array and places the arguments in the array.
  - Note that arguments corresponding to regular parameters are handled in the usual way.

Methods with a Variable Number of Parameters

- Such a method has as the last item on its parameter list a vararg specification of the form:
  ```java
  Type... ArrayName
  ```
  - Note the three dots called an ellipsis that must be included as part of the vararg specification syntax.
- Following the arguments for regular parameters are any number of arguments of the type given in the vararg specification.
  - These arguments are automatically placed in an array.
  - This array can be used in the method definition.
  - Note that a vararg specification allows any number of arguments, including zero.
Method with a Variable Number of Parameters
(Part 1 of 2)

Privacy Leaks with Array Instance Variables

• If an accessor method does return the contents of an array, special care must be taken
  – Just as when an accessor returns a reference to any private object
    public double[] getArray()
    {
      return anArray;//BAD!
    }
  – The example above will result in a privacy leak

Method with a Variable Number of Parameters
(Part 2 of 2)

Privacy Leaks with Array Instance Variables

• The previous accessor method would simply return a reference to the array anArray itself
  • Instead, an accessor method should return a reference to a deep copy of the private array object
    – Below, both a and count are instance variables of the class containing the getArray method
      public double[] getArray()
      {
        double[] temp = new double[count];
        for (int i = 0; i < count; i++)
          temp[i] = a[i];
        return temp
      }
Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

```java
public ClassType[] getArray()
{
    ClassType[] temp = new ClassType[count];
    for (int i = 0; i < count; i++)
        temp[i] = new ClassType(someArray[i]);
    return temp;
}
```

Sorting an Array

- A sort method takes in an array parameter `a`, and rearranges the elements in `a`, so that after the method call is finished, the elements of `a` are sorted in ascending order

- A *selection sort* accomplishes this by using the following algorithm:

  ```
  for (int index = 0; index < count; index++)
      Place the indexth smallest element in a[index]
  ```

Selection Sort (Part 1 of 2)

![Selection Sort, Part 1 of 2](image1)

Selection Sort (Part 2 of 2)

![Selection Sort, Part 2 of 2](image2)
SelectionSort Class (Part 1 of 5)

public class SelectionSort
{
    /**
     * Precondition: count <= a.length;
     * The first count indexed variables have values.
     * Action: Sorts a so that a[0] <= a[1] <= ...
     * <= a[count - 1].
     */
}

SelectionSort Class (Part 2 of 5)

public static void sort(double[] a, int count)
{
    int index, indexOfNextSmallest;
    for (index = 0; index < count - 1; index++)//Place the correct value in a[index]:
        indexOfNextSmallest = indexOfSmallest(index, a, count);
        interchange(index, indexOfNextSmallest, a);//a[0]<=a[1]<=...<=a[index] and these are
        //the smallest of the original array
        //elements. The remaining positions contain
        //the rest of the original array elements.
    }
}

SelectionSort Class (Part 3 of 5)

/**
 * Returns the index of the smallest value among
 * a[startIndex], a[startIndex+1], ...
 * a[numberUsed - 1]
 */

private static int indexOfSmallest(int
    startIndex, double[] a, int count)
{
    double min = a[startIndex];
    int indexOfMin = startIndex;
    int index;
    for (index = startIndex + 1;
        index < count; index++)
        if (a[index] < min)
            {min = a[index];
            indexOfMin = index; //min is smallest of a[startIndex] through
            //a[index]
        }
    return indexOfMin;
}

SelectionSort Class (Part 4 of 5)

for (index = startindex + 1;
    index < count; index++)
    if (a[index] < min)
        {min = a[index];
        indexOfMin = index; //min is smallest of a[startIndex] through
        //a[index]
        }
**SelectionSort Class (Part 5 of 5)**

```java
/**
 * Precondition: i and j are legal indices for the array a.
 * Postcondition: Values of a[i] and a[j] have been interchanged.
 */
private static void interchange(int i, int j, double[] a)
{
    double temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp; //original value of a[i]
}
```

---

**Enumerated Types**

- Starting with version 5.0, Java permits enumerated types
  - An enumerated type is a type in which all the values are given in a (typically) short list
- The definition of an enumerated type is normally placed outside of all methods in the same place that named constants are defined:
  ```java
  enum TypeName {VALUE_1, VALUE_2, ..., VALUE_N};
  ```
  - Note that a value of an enumerated type is a kind of named constant and so, by convention, is spelled with all uppercase letters
  - As with any other type, variables can be declared of an enumerated type

---

**Enumerated Types Example**

- Given the following definition of an enumerated type:
  ```java
  enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
  ```
- A variable of this type can be declared as follows:
  ```java
  WorkDay meetingDay, availableDay;
  ```
- The value of a variable of this type can be set to one of the values listed in the definition of the type, or else to the special value `null`:
  ```java
  meetingDay = WorkDay.THURSDAY;
  availableDay = null;
  ```

---

**Enumerated Types Usage**

- Just like other types, variable of this type can be declared and initialized at the same time:
  ```java
  WorkDay meetingDay = WorkDay.THURSDAY;
  ```
  - Note that the value of an enumerated type must be prefaced with the name of the type
- The value of a variable or constant of an enumerated type can be output using `println`
  - The code:
    ```java
    System.out.println(meetingDay);
    ```
  - Will produce the following output:
    ```java
    THURSDAY
    ```
  - As will the code:
    ```java
    System.out.println(WorkDay.THURSDAY);
    ```
  - Note that the type name `WorkDay` is not output
Enumerated Types Usage

- Although they may look like `String` values, values of an enumerated type are not `String` values.
- However, they can be used for tasks which could be done by `String` values and, in some cases, work better:
  - Using a `String` variable allows the possibility of setting the variable to a nonsense value.
  - Using an enumerated type variable constrains the possible values for that variable.
  - An error message will result if an attempt is made to give an enumerated type variable a value that is not defined for its type.

```java
if (meetingDay == availableDay)
    System.out.println("Meeting will be on schedule.");
if (meetingDay == WorkDay.THURSDAY)
    System.out.println("Long weekend!");
```

Some Methods Included with Every Enumerated Type (Part 1 of 3)

- `equals` method:
  - Returns true if its argument is the same value as the calling value. While it is perfectly legal to use `equals`, it is easier and more common to use `==`.
  - `equals(Value)` is equivalent to `Value == Value`.

- `toString` method:
  - Returns the calling value as a string. This is often invoked automatically. For example, this method is invoked automatically when you output a value of the enumerated type using `System.out.println` or when you concatenate a value of the enumerated type to a string. See Display 6.15 for an example of this automatic invocation.
  - `toString()` returns "MONDAY".

The enumerated type `WorkDay` is defined in Display 6.13.
Some Methods Included with Every Enumerated Type (Part 2 of 3)

**Display 6.14 Some Methods Included with Every Enumerated Type**

```java
public int ordinal()
```

Returns the position of the calling value in the list of enumerated type values. The first position is 0.

**EXAMPLE**

```java
WorkDay.MONDAY.ordinal() returns 0, WorkDay.TUESDAY.ordinal() returns 1, and so forth. The enumerated type WorkDay is defined in Display 6.13.
```

```java
public int compareTo(Any_Value_Of_The_Enumerated_Type)
```

Returns a negative value if the calling object precedes the argument in the list of values, returns 0 if the calling object equals the argument, and returns a positive value if the argument precedes the calling object.

**EXAMPLE**

```java
WorkDay.TUESDAY.compareTo(WorkDay.THURSDAY) returns a negative value. The type WorkDay is defined in Display 6.13.
```

```java
public EnumeratedType[] values()
```

(continued)

The `values` Method

- To get the full potential from an enumerated type, it is often necessary to cycle through all the values of the type
- Every enumerated type is automatically provided with the static method `values()` which provides this ability
  - It returns an array whose elements are the values of the enumerated type given in the order in which the elements are listed in the definition of the enumerated type
  - The base type of the array that is returned is the enumerated type

The Method `values` (Part 1 of 2)

**Display 6.15 The Method values**

```java
import java.util.Scanner;

public class EnumValueDemo {

    enum WorkDay (MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY);

    public static void main(String[] args) {
        WorkDay[] day = WorkDay.values();

        Scanner keyboard = new Scanner(System.in);
        double hours = 0, sum = 0;

        for (int i = 0; i < day.length; i++) {
            System.out.println("Enter hours worked for " + day[i]);
            hours = keyboard.nextDouble();
            sum += hours;
        }

        System.out.println("Total hours worked = " + sum);
    }
}
```

(continued)
The Method values (Part 2 of 2)

### Programming Tip: Enumerated Types in switch Statements

- Enumerated types can be used to control a `switch` statement
  - The `switch` control expression uses a variable of an enumerated type
  - Case labels are the unqualified values of the same enumerated type
- The enumerated type control variable is set by using the static method `valueOf` to convert an input string to a value of the enumerated type
  - The input string must contain all upper case letters, or be converted to all upper case letters using the `toUpperCase` method

---

Enumerated Type in a `switch` Statement (Part 1 of 3)

```java
import java.util.Scanner;

public class EnumSwitchDemo
{
    enum Flavor (VANILLA, CHOCOLATE, STRAWBERRY);
    public static void main(String[] args)
    {
        Flavor favorite = null;
        Scanner keyboard = new Scanner(System.in);
        System.out.println("What is your favorite flavor?");
        String answer = keyboard.nextLine();
        answer = answer.toUpperCase();
        favorite = Flavor.valueOf(answer);
        switch (favorite)
        {
        case VANILLA:
            System.out.println("Classic");
            break;
        case CHOCOLATE:
            System.out.println("Rich");
            break;
        default:
            System.out.println("I but you cold STRAWBERRY.");
            break;
        }
    }
}
```

(continued)
Multidimensional Arrays

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

```java
double[][] table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][][] = new Person[10][100];
```

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column
    ```java
    char[][] a = new char[5][12];
    ```
  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, char)
- In Java, a two-dimensional array, such as `a`, is actually an array of arrays
  - The array `a` contains a reference to a one-dimensional array of size 5 with a base type of `char[]`
  - Each indexed variable (`a[0]`, `a[1]`, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of `char[]`
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions
Using the `length` Instance Variable

```java
char[][] page = new char[30][100];
```

- The instance variable `length` does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable `length` gives the number of first indices (or "rows") in the array
    - `page.length` is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing `length` for that "row" variable
    - `page[0].length` is equal to 100
Ragged Arrays

- Each row in a two-dimensional array need not have the same number of elements
  - Different rows can have different numbers of columns
- An array that has a different number of elements per row it is called a **ragged array**

```java
double[][] a = new double[3][5];
```

- The above line is equivalent to the following:
  ```java
double [][] a;
da = new double[3][]; //Note below
da[0] = new double[5];
da[1] = new double[5];
da[2] = new double[5];
```
  - Note that the second line makes `a` the name of an array with room for 3 entries, each of which can be an array of doubles that can be of any length
  - The next 3 lines each create an array of doubles of size 5

```
double [][] a;
a = new double[3][];
```

- Since the above line does not specify the size of `a[0]`, `a[1]`, or `a[2]`, each could be made a different size instead:
  ```java
  a[0] = new double[5];
a[1] = new double[10];
a[2] = new double[4];
  ```

---

**Multidimensional Array Parameters and Returned Values**

- Methods may have multidimensional array parameters
  - They are specified in a way similar to one-dimensional arrays
  - They use the same number of sets of square brackets as they have dimensions
    ```java
    public void myMethod(int[] [] a)
    {
    . . .
    }
    ```
  - The parameter `a` is a two-dimensional array
Multidimensional Array Parameters and Returned Values

- Methods may have a multidimensional array type as their return type
  - They use the same kind of type specification as for a multidimensional array parameter
    ```java
    public double[][] aMethod()
    {
        ...
    }
    - The method `aMethod` returns an array of `double`

A Grade Book Class

- As an example of using arrays in a program, a class `GradeBook` is used to process quiz scores
- Objects of this class have three instance variables
  - `grade`: a two-dimensional array that records the grade of each student on each quiz
  - `studentAverage`: an array used to record the average quiz score for each student
  - `quizAverage`: an array used to record the average score for each quiz

A Grade Book Class

- The score that student 1 received on quiz number 3 is recorded in `grade[0][2]`
- The average quiz grade for student 2 is recorded in `studentAverage[1]`
- The average score for quiz 3 is recorded in `quizAverage[2]`
- Note the relationship between the three arrays