CMSC 131: Chapter 11 (Supplement)
Arrays

Arrays

Array: A structure used for storing and processing a collection of data all of a common type.

Example: Store a list of 100 integer scores.

- **How to store them?** We could create 100 variables: score0, score1, score2, … but this would be quite tedious.
- **Array**: Allows us to store a collection of items under a single name, and refer to individual items by an **integer index**.
- **Array indices**: Array indices start with 0. So a 100-element array would be indexed from 0 to 99.
- **Array elements**: Are accessed and manipulated just as any variable.

Creating Arrays

An array is a special type of object.

- The notation "[ ]" is Java's way of declaring an identifier to be an array.
- We must invoke "**new**" to allocate storage for each array that we want to use.
- When we allocate a new array, we need to specify **how many** elements we want in the array. This value is put in square brackets.

Example: This declares score to be a 100-element array of **int**'s.

```
int[ ] score = new int[100];


int[ ] score;                          // this is equivalent to the above
score = new int[100];
```

Square Brackets

Don't confuse the three uses of square brackets:

- **Declare** a variable to an **array**:

  ```
  int score[ ];
  ```

  The **base type** of the array is **int**.

- **Specify the size** of a newly created array:

  ```
  score = new int[100];
  ```

- **Access an individual element** of the array:

  ```
  System.out.println( "Last score: " + score[99] );
  ```

Example

Example: Input a list of 4 doubles and print them in reverse order.

```
double[ ] value = new double[4];

for ( int i = 0; i < 4; i++ )
    value[i] = readDouble( );

for ( int i = 3; i >= 0; i-- )
    System.out.println( value[i] );

public static double readDouble( ) {
    return Double.parseDouble(
                    JOptionPane.showInputDialog( "Next value:" ) );
}
```

Array Length

You can determine the number of elements of an array using the **length** instance variable:

```
float[ ] a = new float[5];          // creates a[0] … a[4]
char[ ] b = new char[4];            // creates b[0] … b[3]
int x = a.length;                   // x = 5
int y = b.length;                   // y = 4
```

Since a and b are only references to an array, you can change them.

```
a = new float[6];                   // (can assign to any float array)
int z = a.length;                   // z = 6
```

Array Length

You can access an array's length, but you **cannot change it**:

```
a.length = 40;     // Illegal!  Cannot change length
```

What if I need a bigger array?

Sorry, you must create an **entirely new array**. We will discuss this later.

What do think this does? Outputs 0? Compiler/run-time error?

```
float[ ] c;
c = null;
System.out.println( c.length );
```

Index Out of Bounds

Index out of bounds: If an array is of length m, then valid indices are 0, 1, 2, …, m-1. An attempt to use any other value results in a run-time error: **ArrayIndexOutOfBoundsException**.

Example: Attempt to initialize a 10-element array of doubles:

```
double[ ] list = new double[10];      // valid indices are [0 … 9]
for ( int i = 0; i <= 10; i++ )
    list[i] = 1.0;
```

Fixed:

```
double[ ] list = new double[10];
for ( int i = 0; i < 10; i++ )
    list[i] = 1.0;
```

Better array loop:

```
for ( int i = 0; i < list.length; i++ ) …
```

Alternate Declaration Syntax

Java provides two ways to declare a variable to be an array:

- **Java style**:
  ```
  int[ ] grade;
  ```

- **C and C++ style** (old, but still Java allows):
  ```
  int grade[ ];
  ```

- Java style is preferred (but old C and C++'s programmers sometimes slip into the old style).
  ```
  int[ ] a, b, c;          // a, b, c are all int arrays
  int a[ ], b, c[ ];       // a and c are int arrays, and b is just an int
  ```

Array Initializers

An array can be **initialized when it is declared**. This is done by giving an **initializer list**.

```
int[ ] cutOffs = { 90, 82, 75, 66 };
```

This does a number of things:

- **Counts** the number of elements in the initializer (4 in this case) and automatically invokes new: "**new int[4]**".

- **Initializes** the array values.

- **Returns a reference** to the array (stored in **cutOffs**).

```
int[ ] cutOffs = new int[4];
cutOffs[0] = 90;
cutOffs[1] = 82;
cutOffs[2] = 75;
cutOffs[3] = 66;
```

If not initialized: array elements are set to their **Java default values** (0, 0.0, false, null).

Example

A static method that is given an integer numeric score from 0 to 100, and converts it into a letter grade according to a given set of cut-offs. A grade of 'F' is given if it is less than the lowest score.

```
public static char letterGrade( int numeric ) {
    int[ ] cutOffs = { 90, 82, 75, 66 };
    char[ ] letters = { 'A', 'B', 'C', 'D' };
    for ( int i = 0; i < cutOffs.length; i++ ) {
        if ( numeric >= cutOffs[i] ) {
            return letters[i];
        }
    }
    return 'F';
}
```

Arrays of Characters and Strings

String != char[ ]: In some programming languages, a string is the same as an array of chars. Not in Java.

```
String:          String str = "Hello";
Array of chars:  char[ ] chr = "Hello";      // Illegal! Type mismatch.
                 char[ ] chr = { 'H', 'e', 'l', 'l', 'o' };
```

Other differences:

Operation	String	char[ ]	Result
Get length	str.length( )	chr.length	5
Get char	str.charAt[4]	chr[4]	'o'
Set char	**(not possible)**	chr[0] = 'J';	"Jello"

Converting char[ ] to String: The String constructor is overloaded to allow a char[ ] parameter:

```
String str3 = new String( chr );
```

Array Arguments

Array and array elements: can be passed as parameters to methods, just like primitive types and object references.

Passing a Single Element: A single element of an array can be passed to any method as if it were a **simple variable**.

Example: Suppose **fooBar(z)** expects an argument z of type **double**.

```
double[ ] a = new double[10];
int i = …
// … initializations omitted
fooBar( a[3]  );    // passes the value of a[3] to fooBar
fooBar( a[i] );     // evaluates a[i] and passes the value to fooBar
```

Passing an Entire Array: You can pass an entire array as an argument. As with objects, this just passes the **reference**.

- If the method expects an array of type double, then you can pass it an array of doubles **of any size**.

- **Promotion does not apply**. E.g., you cannot pass it an array of int's.

Array Argument Example 1

Example: A static method, **printDoubles**, that is given an array of doubles and prints each, one per line.

```
public static void printDoubles( double[ ] v ) {
    System.out.println( "Array contents:" );
    for ( int i = 0; i < v.length; i++ )
        System.out.println( "  [" + i + "]=" + v[i] + " " );
}

double[ ] height = { 6.42, 7.10, 4.83 };
double[ ] weight = { 122, 170.5 };
printDoubles( height );
printDoubles( weight );
```

Array Argument Example 2

Example: A static method, **squareValues**, that is given an array of doubles and replaces each value with its square.

```
public static void squareValues( double[ ] v ) {
    for ( int i = 0; i < v.length; i++ )
        v[i] = v[i] * v[i];
}

double[ ] height = { 6.42, 7.10, 4.83 };

squareValues( height );
printDoubles( height );
```

Pass by Value: Only the **array reference**, height, is passed. This reference cannot be changed, but the **array contents** can be.

Array Argument Example 3

Example: A static method, **doubleSize**, that is given an array b of char's, creates a new array b2 of **twice the size** of the original. It copies the values of b to b2, and sets the remainder to ' '. It returns b2.

```
public static char[ ] doubleSize( char[ ] b ) {
    char[ ] b2 = new char[ 2 * b.length ];
    for ( int i = 0; i < b.length; i++ )
        b2[i] = b[i];
    for ( int i = b.length; i < b2.length; i++ )
        b2[i] = ' ';
    return b2;
}
```

Usage:

```
char[ ] buffer = { 'J', 'a', 'v', 'a' };
char[ ] newBuffer = doubleSize( buffer );
```

(More to Come)