

Arrays & ArrayList

I. Simple vs. Structured Data Types

A) Recall: Simple data types, such as integers and floats, cannot have an individual element broken down any further.

Structured data types - a collection of components where individual elements can be accessed.

ex.) an array and classes

II. One-Dimensional Arrays (13.4) - array: a sequence of values of the same type

A. 1) Introduction:

ex.1) Read and store 10000 items:

method 1: a, b, c , etc.

or

method 2a: x_0, x_1, x_2 , etc. or method 2b: x_0, x_1, x_2 , etc.

or

method 3: $x[0], x[1], x[3]$, etc.

Note: This last notation is how Java does a list called an **array** in one dimension. It is a structured collection of elements.

Important: The item in the brackets is called the index (position in the list)
The name of the array is the identifier in front of the brackets.

ex.2) **new int[10]**; is an object of ten integer memory locations called elements
int[] studentID; is a reference to the object integer array

One element of the `studentID` array is `studentID[5]`

Does it have to have the value of 5 stored in it? NO!

(The five is its position in the list but its position is not the fifth one!) Why?

The numbering of the position (index) in the list starts with zero, then 1, 2, 3 etc.

ex.3) `x[y]` is an array of `y` elements called the `x` array where `y` must be a non-negative integer, and `x` is of any variable of primitive data type or is an object variable of a class type

- 2) Constructing an array object:
ex.1) **new double**[10]

Creating a reference to the array:

double[] data = **new double**[10]; // note the bracket and not a ()

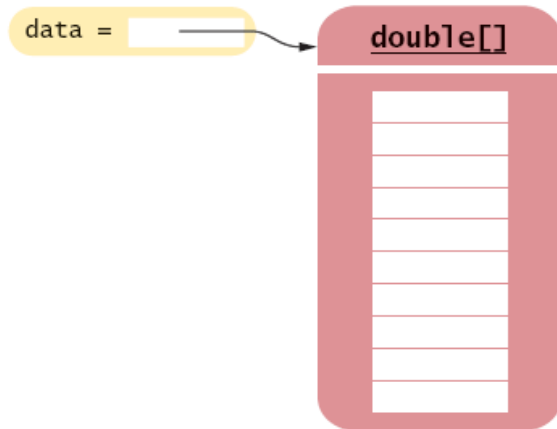


Figure 1 An Array Reference and an Array

When array is created, all values are initialized depending on array type:

Numbers: 0

Boolean: false

Object References: null

Declaring an array continued:

ex.2) **float**[] studentID = **new float**[2200]; // index range: 0-2199

int[] actScore = **new int**[100]; // index range: 0-99

boolean[] example2 = **new boolean**[3]; // index range: 0-2

Fish[] mbs = **new Fish**[5]; // index range: 0-4

- 3) a) Assigning values to an individual array component (element)

ex.1) **double**[] data = **new double**[4];

// declaration of array example of 4 components

data[0] = 100.0;

data[1] = 25.2;

data[2] = 29.95;

data[3] = -5.5;

Use [] to access an element: `data[2] = 29.95; // really the third item`

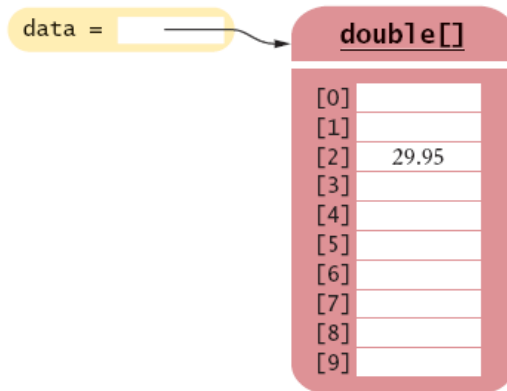


Figure 2 Storing a Value in an Array

Using the value stored:

ex.1) `System.out.println("The value of this data item is " + data[4]);`

b) Initializing in the declaration (a shortcut - no **new** needed)

i) `double[4] example = {100.0, 25.2, -0.1, -5.5}; // notice the braces`

or

ii) `double[] example = {100.0, 25.2, -0.1, -5.5};`

Note: The compiler determines the size of the array based on the number of values listed.

c) Initializing an array using a loop

ex.1)

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

d) Uninitialized array

ex.1)

```
double[ ] data;
data[0] = 5.1; // an error why?
```

Answer: the array was never created only the reference to it was.

4) Subarray processing

ex.1) `float[] studentID = new float[2200];`

Maybe only 2150 elements are used out of the total of 2200.

5) Parallel arrays - Two or more arrays that are accessed by using the same index.

ex.1) `float[] seniors = new float[500];`

`int[] ranking = new int[500];`

`for (number = 0; number < 500; number++)`

`System.out.println(seniors[number] + ranking[number]);`

Note: The arrays do not have to be the same type, they only have to use the *same index* to be parallel arrays!

6) Passing arrays as parameters in methods

ex.1)

public class Example

{

public static void main(String[] args)

{

double[] array1 = {1.1, 2.2, 3.3, 4.4}; // *declarations and initializations*

double[] array2 = {-6.6, -5.5, -4.4, -3.3, -2.2, -1.1};

example1(array1); // *call to the method example1*

example1(array2); // *second call to the method example1*

}

void static example1(**double**[] anArray)

{

int index;

for (index = 0; index < anArray.length; index++) // see next page for length

System.out.println(anArray[index]);

}

}

Note : The reference to the array object is passed, just like for any object.

- 7) Passing an individual array component as a parameter in methods
ex.1)

```
public class Example
{
    public static void main(String[ ] args)
    {
        double[ ] array1 = {1.1, 2.2, 3.3, 4.4}; // declarations and initializations
        example1(array1[0]);                // call to the method example1
        example1(array1[1]);                // second call to the method example1
    }

    void static example1(double x)
    {
        System.out.println(x);
    }
}
```

Output:

1.1
2.2

- 8) Automatic initializing

```
double[ ] data1 = new double[10]; // all 10 components are set to 0.0
```

```
int[ ] data2 = new int[10]; // all 10 components are set to 0
```

```
char[ ] data3 = new char[10]; // all 10 components are set to null
```

```
String[ ] data4 = new String[10]; // all 10 components are set to null
```

B. Getting array length: `data.length` or `x.length`

Note: This is not a method! It (`length`) is an instance variable of the array object.

Note: Index values range from 0 to `length - 1`

ex.1) initializing again

```
double[ ] data1 = new double[10];  
for (int i = 0; i < data1.length; i++)  
    data1[i] = i;
```

Accessing a nonexistent element results in a bounds error.

(usually called an *out of bounds* error)

ex.1)

```
double[ ] data1 = new double[10];  
data1[10] = 29.95; // ERROR
```

C. 1) Copying an array

ex.1)

```
double[ ] data1 = new double[10];  
double[ ] data2 = data1;
```

`data2` now points to the same object as `data1`

if `data1[4] = 29.5`; then `data2[4]` is 29.5

2) if you want `data2` to be a copy of `data1` then clone it:

ex.2)

```
double[ ] data1 = new double[10];  
double[ ] data2 = (double[ ])data1.clone( );
```

(`double[]`) is a cast because the clone method is for the Object type

if `data1[4] = 29.5`; then `data2[4]` is 0 (or some other value)

D. Limitation of Arrays: Arrays have fixed length

If the array is full and you need more space, you can grow the array:

1. Create a new, larger array.

```
double[ ] newData = new double[2 * data.length];
```

2. Copy all elements into the new array

```
System.arraycopy(data, 0, newData, 0, data.length);
```

3. Store the reference to the new array in the array variable

```
data = newData;
```

Too bad there is not an easier way of doing this...

- E. An array can have objects as its components:
ex.1)

```
public class A
{
    String name;
    int numberLetters;
    public A(String n)
    {
        name = n;
        numberLetters = name.length( );
    }
}

public class B
{
    A[ ] list;
    public B(int num, String word1, String word2, String word3, String word4)
    {
        list = new A[num];           // note the brakets, this is not a constructor call
        list[0] = new A(word1);     // this is a constructor call
        list[1] = new A(word2);
        list[2] = new A(word3);
        list[3] = new A(word4);
    }
}
```

Client file:

```
B me = new B(4, "Bob", "Robert", "Bobby", "Rob");
B you = new B(2, "Jon", "Johnny", "", "");
```

One memory location containing an A object:

```
list[0]
name = Bob           (Note: B.list[0].A.name)
numberLetters = 3
```

AP Programming - Chapter 13 Lecture

III. Array Lists (13.1) - the ArrayList class manages a sequence of objects

Note: The ArrayList class is part of the **java.util** package

use: **import java.util.ArrayList;**

A) Declaring

ex.1) ArrayList accounts = **new** ArrayList();

B) Properties and methods needed for AP test:

- Can grow and shrink as needed
- ArrayList class supplies methods for many common tasks, such as inserting and removing elements
- size() method yields number of elements
- The ArrayList class is a generic class:

Java version 5.0: ArrayList<T> collects objects of type T :

where T is called a type parameter

ex.1)

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>( );
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

1) get(index) method - retrieves an array element at position index

Note: the index starts at 0

ex.1)

```
ArrayList accounts = new ArrayList( );
BankAccount anAccount = accounts.get(2);
// gets the third element of the array list
```

Note: Bounds error if index is out of range

Most common bounds error:

```
int i = accounts.size( );
```

```
anAccount = accounts.get(i); // Error - legal index values are: 0. . . i - 1
```

2) set(index, object) method - overwrites an existing value

ex.1)

```
ArrayList accounts = new ArrayList( );
BankAccount anAccount = new BankAccount( );
accounts.set(2, anAccount);
// adds a new object value (anAccount) at the index of 2
// and overwrites whatever was there
```

or

```
accounts.set(2, new BankAccount(100, 98765));
```

AP Programming - Chapter 13 Lecture

- 3) a) `add(object)` method - adds an object value to the end of the list and adjusts the size of the array

ex.1)

```
ArrayList accounts = new ArrayList( );  
BankAccount anAccount = new BankAccount( );  
accounts.add(anAccount);  
    // adds a new object value (anAccount) to the end of the list
```

- b) `add(index, object)` method - adds an object value at the position index and moves all objects back a position (by adding 1 to their index) and adjusts the size of the array

ex.1)

```
ArrayList accounts = new ArrayList( );  
BankAccount anAccount = new BankAccount( );  
accounts.add(3,anAccount);  
    // adds a new object value (anAccount) at position 3  
    // (i.e. the 4th element)
```

- 4) `remove(index)` method - removes an existing value at position index and moves all objects up a position (by subtracting 1 to their index) and adjusts the size of the array

ex.1)

```
ArrayList accounts = new ArrayList( );  
accounts.remove(2);  
    // removes an object value at the index of 2  
    // (i.e. the 3rd element)
```

- 5) `size()` method - returns the number of elements in the ArrayList

ex.1)

```
ArrayList accounts = new ArrayList( );  
int x = accounts.size( );  
    // returns the number of elements in the ArrayList and  
    // stores it in the variable x
```

IV. Arrays vs ArrayLists:

Arrays

random-access, linear data structure
fixed size once created
can contain objects and primitives
must declare element type
safe: run-time bounds checking

```
Fish[ ] myArray = new Fish[15];
```

```
myArray[index] = new Fish(loc);  
(no way to easily add – arraycopy)
```

```
Fish f = myArray[index];
```

(no way to easily remove – arraycopy)

```
for(int k = 0; k < myArray.length; k++)  
    System.out.println(myArray[k]);
```

Recall: To find the length of a String use a method in the String class called length().

ex.1) String ex = "Cat";

```
int x = ex.length( ); // x will have a value of 3
```

ArrayLists

random-access, linear data structure
dynamic size; grows automatically
can only contain objects
element type is Object
safe: run-time bounds checking

```
ArrayList myList = new ArrayList( );
```

```
myList.add(new Fish(loc)); or  
myList.add(3, Fish(loc)); or  
myList.set(index, new Fish(loc));
```

```
Fish f = (Fish)myList.get(index);
```

```
myList.remove(index);
```

```
for(int k = 0; k < myList.size( ); k++ )  
    System.out.println(myList.get(k));
```

Methods in AP Subset:

```
boolean add(Object obj)
```

```
void add(int index, Object obj)
```

```
Object get(int index)
```

```
Object set(int index, Object obj)
```

```
Object remove(int index)
```

```
boolean remove(Object obj)
```

Note: *MBS only*

```
int size( )
```

```
Iterator iterator( ) - AB topic
```

```
Iterator listIterator( ) - AB topic
```

V. Wrapper Classes

ArrayLists can only contain objects. If you want to store primitive numbers in an ArrayList you would need to convert them to an object and there are Wrapper classes that can do that. (Also if you are going to use the Object class methods equals() or compareTo() the data must be an object.)

There are wrapper classes for all eight primitive types:
(Note the Integer and Character ones.)

| Primitive Type | Wrapper Class |
|----------------|---------------|
| byte | Byte |
| boolean | Boolean |
| char | Character |
| double | Double |
| float | Float |
| int | Integer |
| long | Long |
| short | Short |

ex.1)

```
ArrayList num = new ArrayList( );  
double x = 5.1;  
Double wrappedNum = new Double(x);  
num.add(wrappedNum); // the new object, 5.1, has been added to num ArrayList  
Double wrapperEx = (Double)num.get(0);  
// now unwrap it:  
double y = wrapperEx.doubleValue( ); // doubleValue is unWrapper method
```

or

```
ArrayList num = new ArrayList( );  
num.add(new Double(5.1)); // the new object, 5.1, has been added to num ArrayList  
Double wrapperEx = (Double)num.get(0);  
// now unwrap it:  
double y = wrapperEx.doubleValue( ); // doubleValue is unWrapper method
```

VI. Wrapper Classes - a shortcut in Java version 5.0

Note: You can not do: `ArrayList<double>` but you can do: `ArrayList<Double>`

A) auto-boxing (really auto-wrapper)

ex.1) `Double x = 5.1;` // automatically converts the 5.1 to an object (in Java 5)
and
`double y = x;` // automatically converts the object 5.1 to the double 5.1

Controversy Over Boxing and Unboxing

Not all software developers are happy with Sun's decision to add boxing and unboxing to the language. The fact that an `ArrayList<Integer>` can be manipulated almost as if it were an `ArrayList<int>` can simplify code and everyone agrees that simplification is good. But the disagreement comes from the fact that it is almost like an `ArrayList<int>`. The argument is that "almost" isn't good enough. The fact that it comes close means that you are likely to use it and eventually come to count on it. That can prove disastrous when "almost" isn't "always."

For example, suppose that someone told you to use a device that is almost like a potholder. In most cases, it will protect your hand from heat. So you start using it and you might be nervous at first, but then you find that it seems to work just fine. And then one day you're surprised to find that a small area isn't behaving like a potholder and you get burned. You can think of similar scenarios with aircraft landing gear that almost works or vests that are almost bullet-proof.

For example, consider the following code:

```
int n = 420;
ArrayList<Integer> list = new ArrayList<Integer>( );
list.add(n);
list.add(n);
if (list.get(0) == list.get(1))
    System.out.println("equal");
else
    System.out.println("unequal");
```

It's difficult to know exactly what this code will do. If we think of the `ArrayList<Integer>` as being "almost" like an `ArrayList<int>`, we would be inclined to think that the code would print the message that the two values are equal. In fact, there is no guarantee as to what it will do. In the current release of Java, it prints the message "unequal".

Remember that testing for object equality is not as simple as testing for equality of primitive data. Two Strings might store the same text but not be the same object, which is why we call the `equals` method to compare Strings. The same applies here. The two list elements might store the same int but not be the same object. The code above prints "unequal" in the current release of Java because the program creates two different Integer objects that each store the value 420. But if we change the value from 420 to 42, suddenly the program prints that the two values are equal. The Java Language Specification guarantees that this code will work for any value of n between -128 and 127, but provides no guarantee as to how the code will behave for other values of n. For those other values it could print either message. And we have no guarantee of how this might change from one implementation of Java to another. It might be that in the next release of Java, the code will print "equals" for 420 but not for a value like 420000.

Some have argued that because boxing and unboxing cover up what is happening underneath, that it is better not to use them at all. Boxing and unboxing don't necessarily simplify anything if they work only "sometimes" because you have to be able to understand the cases where they don't work.

Here's an example:

```
Integer x = 42;
Integer y = 42;
System.out.println(x == y);
```

This code will print "true" (they are the same object). But if you say:

```
Integer x = new Integer(42);
Integer y = new Integer(42);
System.out.println(x == y);
```

This code will print "false" (they are different objects).

The Java language specification says that Java will cache certain commonly used values when it autoboxes.

--Stuart Reges, Senior Lecturer, University of Washington

<http://www.cs.washington.edu/homes/reges>

* R. Greenlee AP Programming Java 2008 Wheaton Warrenville South High School, Wheaton IL. *

VI. **for** loop shortcut in Java version 5.0

ex.1)

Traditional method of traversing all elements of a list and summing them:

```
double [ ] data = { . . . };  
double sum = 0;  
for (int i = 0; i < data.length; i++)  
{  
    double e = data[i];  
    sum = sum + e;  
}
```

version 5.0 shortcut:

```
double [ ] data = { . . . };  
double sum = 0;  
for (double e : data) // You should read this loop as "for each e in data"  
{  
    sum = sum + e;  
}
```

ex.2) Works for ArrayLists too:

Traditional **for** loop:

```
    double sum = 0;  
for (int i = 0; i < accounts.size( ); i++)  
{  
    BankAccount a = accounts.get(i);  
    sum = sum + a.getBalance( );  
}
```

version 5.0 shortcut:

```
ArrayList<BankAccount> accounts = . . . ;  
double sum = 0;  
for (BankAccount a : accounts)  
{  
    sum = sum + a.getBalance( );  
}
```

Summary:

```
for (Type variable : collection)  
    statement
```

Purpose:

To execute a loop for each element in the collection. In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

Example:

```
for (double e : data)  
    sum = sum + e;
```


C. Two-Dimensioned Arrays / Initialization continued

The data: 2 4 6
 3 5 7

2) Using an assignment statement for each element (usually not practical)

```
int[ ][ ] example = new int[2][3];
example[0][0] = 2;
example[0][1] = 4;
example[0][2] = 6;
example[1][0] = 3;
example[1][1] = 5;
example[1][2] = 7;
```

3) Using **for** loops (very common)

```
int[ ][ ] example = new int[2][3];           // declaration
int i, j;
for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
        example[i][j] = Stdin.readInt(); // recall: shortcut read from import hsa.Stdin
}
```

D) Outputting a table: (almost the same as inputting the data)

```
int[ ][ ] example = new int[2][3];           // declaration
int i, j;
for (i = 0; i < 2; i++)
{
    for (j = 0; j < 3; j++)
        System.out.print(example[i][j] + " ");
    System.out.println();                     // needed to move to the next row
}
```

E) Summing (accumulating) row or column values

- 1) When summing a row of values from a table the row position will be a constant and the column position will go from 0 to column # -1

ex. a) summing (accumulating or totaling) the values in row 6

```
int [ ] [ ] example = new int[20] [30];    // declaration
int j ;
total = 0 ;
for (j = 0; j < 30; j++)
    total = total + example[5] [j] ;
```

ex. b) summing (accumulating or totaling) the values in each of the rows and then finding the average of each of the rows

```
int [ ] [ ] example = new int[20] [30];
int row_ave[20];
int i, j ;
for (i = 0; i < 20; i++)
{
    total = 0 ;
    for (j = 0; j < 30; j++)
        total = total + example[i] [j] ;
    row_ave[i] = total/30 ;
}
```

E) Summing (accumulating) row or column values continued

2) When summing a column of values from a table the column position will be a constant and the row position will go from 0 to row # minus 1

ex. a) summing (accumulating or totaling) the values in column 11

```
int[ ][ ] example = new int[20] [30];    // declaration
int i ;
total = 0 ;
for (i = 0; i < 20; i++)
    total = total + example[i][10] ;
```

ex. b) summing (accumulating or totaling) the values in each of the columns and then finding the average of each of the columns

```
int[ ][ ] example = new int[20] [30];
int column_ave[30] ;
int i ;
for (j = 0; j < 30; j++)
{
    total = 0 ;
    for (i = 0; i < 20; i++)
        total = total + example[i][j] ;
    column_ave[j] = total/20 ;
}
```

F) Passing Two-Dimensioned Array parameters to a method:

ex.1) functionEx(ex, rowSize); // method call

```
void functionEx(int[ ][3] ex, int rowSize) // method header
    // rowSize unnecessary but common for use in later possible loops
```

Note: In the method call no brackets are used

Note: Just as in one-dimensioned arrays the row index is not included in the method header. If it is supplied it is really ignored by the computer. The column number must be included! The reason is that the base address (starting location of the first element) of the array is passed to the method but the computer needs to know how many columns there are so it knows when the next row starts.

Optional: `ex[][3]` leads to the idea that a two-dimensioned array can be thought of as an array of arrays.

VIII. Multi-dimensioned arrays

A) Arrays do not have to be limited to just one or two dimensions but can have an unlimited number of dimensions (restricted by the amount of memory)

e

ex.) a page in a book:

```
ex[r] [c]           //row, column
ex[p] [r] [c]      //page, row, column
ex[b] [p] [r] [c]  //book, page, row, column
ex[s] [b] [p] [r] [c] //shelf, book, page, row, column
ex[c] [s] [b] [p] [r] [c] //bookcase, shelf, book, page, row, column
ex[f] [c] [s] [b] [p] [r] [c] //floor, bookcase, shelf, book, page, row, column
```

IX. Random Numbers review from Chapter 6 needed for this unit's Take-Home Test

- A) Random Numbers using the static **random()** method found in the Math class (new AP topic in 2006-2007)

Math.random() produces a *pseudorandom* random decimal number between 0 (inclusive) and n (exclusive)

The following line would generate a random number between *Big* number and *Small* number inclusive:

(int)(Math.random()*(*Big* - *Small* + 1)) + *Small* or
(int)(Math.random()*(*Big* - *Small* + 1) + *Small*)

- ex.1) Generate a random number between 5 and 10 inclusive

Note 1: *Big* is the larger number, 10 in this example

Note 2: *Small* is the smaller number, 5 in this example

Note 3: the formula needs a typecasting of **int**

Answer: **(int)**(Math.random()*(10 - 5 + 1) + 5) or **(int)**(Math.random()*(6) + 5)

- ex.2) generate a random number between 15 and 25 inclusive and store in x

Answer: **int** x = **(int)**(Math.random()*(11) + 15);

- ex.3) **(int)**(Math.random()*(100) + 10) will produce a random number between what two numbers?

Answer: between 10 and 109 inclusive (work: 100 = last - 10 + 1)

- B) Random Numbers using the Random Class (AP topic)

The Random Class (found in **import** java.util.Random;) has two methods in it that can be used to generate *pseudorandom* random numbers:

nextInt(n) - a random integer between 0 (inclusive) and n (exclusive)

nextDouble() - a random floating-point number between 0 (inclusive) and 1 (exclusive)

The following two lines would generate a random number between *Big* number and *Small* number inclusive:

Random generator = **new** Random();

int x = generator.nextInt(*Big* - *Small* + 1) + *Small*;

- ex.1) generate a random number between 1 and 5 inclusive and store in x:

Random die = **new** Random();

int x = die.nextInt(6) + 1

Chapter 13 Terminology:

structured data type - A collection of components whose organization is characterized by the method used to access individual components.

array (one-dimensional array) - A structured collection of components, all of the same data type, that are accessed by relative position within the collection.

index - Used to specify the component's position within the array.

out-of-bounds - An index value that, in C++, is either less than zero or greater than the array size minus one.

subarray - Refers to working with only that portion of an array that contains meaningful data values.

parallel arrays - A structure in which several arrays are accessed by the same index.

base address - The memory address of the first element of the array.

two-dimensional array - A collection of components, all of the same type structured in two dimensions.

random access - Accessing elements in an arbitrary, rather than sequential, order.