Arrays

Variables give us the ability to store individual numbers, strings, boolean values, or object addresses, but for many tasks we have to deal with dozens, hundreds, or thousands of such values. Instead of creating thousands of individual variables, we can instead use an *array* structure to hold a list of values that are all of the same type. Arrays have built-in numbering and we can use this to easily process values that live in an array using a for loop.

Array Structure

Suppose that we have a list of integer grades that we have to find the average for, and then print. If we don't know about arrays, the best we could do is create a variable for each grade, and name them grade0, grade1, etc. Then every task we want to do with this grades, we have to manually do line by line:

```
int grade0 = 67;
int grade1 = 71;
int grade2 = 99;
// ... do grade3 through grade8
int grade9 = 86;

double average = 0;

average = average + grade0;
average = average + grade1;
// ... do grade2 through grade8
average = average + grade9;
average = average / 10;

System.out.println("Grades:");
System.out.println(grade0);
System.out.println(grade1);
// ... do grade2 through grade8
System.out.println(grade9);
```

Note that even in writing out this much, I was too lazy to write every line, I just said things like "do grade3 through grade8" to indicate that we want to do the same thing to the variable grade followed by a number that changes.

An array gives us a way to number variables in a list, where the number is a separate part that we can programmatically change in order to work our way through the list.

At the hardware level, to create an array we would set out a sequence of rows in Main Memory to hold our list of values – each row in the array acts like a separate variable for storing a value, but the array lets us treat the whole array as one object when we need to.

In Java, arrays are always stored on the heap, just like objects, and the array variable itself is assigned the memory address of the first of these rows. Each subsequent value in the array will be stored on a new row, so we can think of each position in the array as an offset from that first row. The location of the first value is offset by 0, the next value is offset by 1, etc.

We use these numbers, indicating how many rows down to move in the array, as the array *index*, which is always an integer value (you can't move down half a row in memory). We will use square braces to surround the index, at the end of the name of the array. Since we start on the top row of the array, with no offset, indices[1] always start at 0.

The list of values stored in the array are its *elements*.

To create an array of any type, we have to say how many rows in memory to reserve for the whole list of elements. We also use square braces around this number when we create the array.

```
// declaring an array variable (no actual array)
type[] arrayVariableName;2
// initializing the array (creating it on the heap)
arrayVariableName = new type[sizeOfArray];

// putting a value into an array at an index
arrayVariableName[index] = valueForElement;
```

Let's rewrite the average code above, using array syntax:

```
1  // declare an array of ints called grades
2  // and set it up to be size 10
3       // "int array grades is size ten"
4       // "grade is an int array, size 10"
5       // "grade is an array of 10 ints"
```

---

[1] This is the plural of index.

[2] It is also legal to say

```
type arrayVariableName[]
```

but using `type[]` emphasizes that "array of type" is the type of the variable, which is a more standard way to think of it in java.

```
 6  int[] grade;  // array variable, no array object yet
 7  grade = new int[10]; // create array object on heap
 8
 9  grade[0] = 67; // put 67 in 0th row of the grades array
10  grade[1] = 71; // put 71 on 1th row of the grades array
11  grade[2] = 99; // put 71 on 2th row of the grades array
12  // ... do grade[3] through grade[8]
13  grade[9] = 86; // put 86 on 9th row of the grades array
14
15  double average = 0;
16  // add the value from the 0th row of the array
17  average = average + grade[0];
18  // add the value from the 1th row of the array
19  average = average + grade[1];
20  // ... do grade[2] through grade[8]
21  average = average + grade[9];
22  average = average / 10;
23
24  System.out.println("Grades:");
25  System.out.println(grade[0]);
26  System.out.println(grade[1]);
27  // ... do grade[2] through grade[8]
28  System.out.println(grade[9]);
```

Let's look at what's happening under the hood. In line 6 we are declaring the array variable, but we have not yet created the array itself, so all we have so far in memory is

`int[] grade;`

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
grades	51	null
	...	

In line 7 we use new to actually create the array on the heap, and store the memory address of the array in the grades variable.

grade = new int[10];

variable	*MM*	*value*

[stackframe: main (args)]		
args	50	null
grades	51	MM100
	…	
[heap]		
Index: 0	100	0
1	101	0
2	102	0
3	103	0
4	104	0
5	105	0
6	106	0
7	107	0
8	108	0
9	109	0
	…	

Rows 9, 10, 11, and13 use the elements of the array as variables to store values (the code also assumes that we fill in the other rows, but let's just look at what these lines would do:

```
grade[0] = 67; // put 67 in 0th row of the grades array
grade[1] = 71; // put 71 on 1th row of the grades array
grade[2] = 99; // put 71 on 2th row of the grades array
grade[9] = 86; // put 86 on 9th row of the grades array
```

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
grades	51	MM100
	…	
[heap]		
Index: 0	100	67
1	101	71
2	102	99
3	103	0
4	104	0
5	105	0
6	106	0
7	107	0
8	108	0
9	109	86
	…	

All we have done so far is switch to array syntax, we haven't yet made use of the strength of arrays to simplify this code. We need a little more understanding first.

Array Index

Notice the somewhat odd wording I used, I called grade[0] the 0th row and grade[1] the 1th row. Talking about the "first" row of an array can be ambiguous: do I mean the very first one, which is at offset 0, or the first one as in the one numbered "1" which is actually the second row in the array, offset by 1? Because of this, people have come up with different ways to refer to the rows such as 0th and 1th [3] or always saying "index 0" and "index 1" or "offset 0" and "offset 1" etc.

When talking about code, we generally just say the name of the array followed by "sub" then the index so "grade sub 0" to mean grade[0] or "grade sub 1" to mean grade[1] or even leave out the sub and just say "grade 0" to mean grade[0] and "grade 1" to mean grade[1]

Because index always starts at 0, the last index in the array is always 1 less than the size we created the array at. We initialized grade with size 10, so grades runs from index 0 to index 9 – it takes up 10 rows, but those rows are numbered 0..9.

I was able to do math using the elements of the array and print the elements of the array, just like any other integers. At the moment variable names like grade[3] look strange because they are new, but they really are just variable names, and I can do anything with grade[3], an integer that lives in an array structure at index 3, that I could have done with an integer variable grade3, an integer that lives in an individual variable.

Using Values in an Array

So far, changing to array format hasn't bought us anything, we're still manually moving through just as many lines of code. This will change when we start using for loops with arrays. Let's look at a few more array examples first

```
// declare variable rents as an array that holds
// 10 double values, indices 0 to 99
double[] rent = new double[100];
// put the value 1560 at index 0 in the rent array
rent[0] = 1560.00;
// ... fill in rent[1] to rent[89]
// put the value 1290 at index 99 in the rent array
rent[99] = 1290.00;
```

[3] Or aloud, pronouncing 1st as "wunst" to indicate index 1.

```
// using a variable to hold the index
int myAptNum = 67;
rent[myAptNum] = 1350.00;
// same as
rent[67] = 1350.00;

// my rent is stored on row 22
// and I have to pay $132 in utilities
double monthTotal = rent[22] + 132.00;

// declare variable petNames as an array
// that holds 15 string values
// indices 0 to 14
String[] petNames = new String[15];
// put the value “Fluffy” at index 0 of the petNames array
petNames[0] = "Fluffy";
// . . .  fill in petNames[1] to petNames[13]
// put the value “Spot” at index 14 of the petNames array
petNames[14] = "Spot";

// my pet is on 4th row (index 3) and
// I am printing about a vet appointment
int x = 4;
System.out.println(petNames[x - 1] + " has a vet appointment on
     May 3.");

// declare variable catCanFly as an array
// that holds 5 boolean values
// indices 0 to 4
boolean[] catCanFly = new boolean[5];
// put the value true at index 0 of the catCanFly array
catCanFly[0] = true;
// . . .  fill in catCanFly[1] to catCanFly[3]
// put the value false at index 4 of the catCanFly array
catCanFly[4] = false;

// get which row their cat is on and then
// advise them on what activity to do
```

```
System.out.println("what index is your cat at in the array?")
int catIndex = scan.nextInt();

if (catCanFly[catIndex]) {
    System.out.println("Go ahead and take your cat flying");
} else {
    System.out.println("Just play with your cat on the ground");
}
```

Since the array index is always an int, no matter what type is in the array, we can use literal int values, or anything at evaluates to an integer, like an int variable or an expression, in the square braces to indicate which row in the array we want, which we did in the above code.

CS Numbers vs Human Numbers

At the end of that code, we asked the user for the array index to use. Most of the time, users would be confused by using "computer scientist numbers" that start at 0, so we would adjust between these and "human numbers" that start at 1 when talking to a user:

```
// get which row their cat is on
System.out.println("which cat (1 to 5)?");
int catIndex = scan.nextInt();

// they entered a human number
// subtract 1 to make it a correct array index
catIndex = catIndex - 1;

// advise them on what activity to do
if (catCanFly[catIndex]) {
    System.out.println("Go ahead and take your cat flying");
} else {
    System.out.println("Just play with your cat on the ground");
}
```

Array Initialization

Like instance variables, the contents of arrays are initialized with the usual default values: 0 for numbers, false for booleans, null for everything else.

If we know the values we want for an array when it is created, we can do a *special initialization* to put them all into place at once. We do this by putting them into a comma separated list inside curly braces:

```
// special initialization
String[] zooAnimals = {"elephant", "zebra", "hippo", "penguin",
     "aardvark"};

// same as saying
String[] zooAnimals = new String[5];
zooAnimals[0] = "elephant";
zooAnimals[1] = "zebra";
zooAnimals[2] = "hippo";
zooAnimals[3] = "penguin";
zooAnimals[4] = "aardvark";
```

Array Size

Array structures are static, which means that whatever size they were when we created the array, they stay that size and cannot change. Trying to put data outside the size of the array is a problem. No array has negative indices, and remember that valid indices in the array always go up to one less than the size.

Suppose we used an invalid index in an array. This causes a runtime error that causes the program to halt, an ArrayIndexOutOfBoundsException.

We know it is never safe to use a negative number as the index, but to avoid the index being too big, we need to know how big the array is. Arrays in java know their own length, which we can get to using the .operator. Although this makes it look like the length is an instance variable, remember that it cannot be changed.

```
// declare variable petNames as an array
// that holds 8 string values
// indices 0 to 7
// so petNames.length is 8
String[] petNames = new String[8];
petNames[0] = “Muffy”;
petNames[1] = “Fluffy”;
petNames[2] = “Clyde”;
```

```
System.out.println("there are " + petNames.length + " pet names
     in our list");

// put “Rover” in the last position in petNames
// which must be its size minus 1
petNames[petNames.length - 1] = "Rover";
```

For the purpose of examples, I will put the length at the end of the array when showing examples, so at the end of this code:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
petNames	51	MM100
	…	
[heap]		
Index: 0	100	“Muffy”
1	101	“Fluffy”
2	102	“Clyde”
3	103	null
4	104	null
5	105	null
6	106	null
7	107	“Rover”
length	108	8
	…	

For Loops for Arrays

The real power of arrays comes from the fact that we can use a variable in the square braces, and use a for loop to change that variable. As a result, we can write a for loop to do the same thing to each element of the array. We want to end up writing code like

```
// add up all the grades in the array
for (int i = 0; i < grade.length; i++) {
    total = total + grade[i];
}
```

But let’s work our way up to that, to make sure we understand what we’re doing.

First, let’s revisit doing all the steps of an average manually, using array syntax:

```
// new list of grades using special initialization
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49};
double average = 0;
double total = 0;

// go through each grade in the array, adding it onto the total
total = total + grade[0];
total = total + grade[1];
total = total + grade[2];
total = total + grade[3];
total = total + grade[4];
total = total + grade[5];
total = total + grade[6];
total = total + grade[7];
total = total + grade[8];
total = total + grade[9];

// math for the average
average = total / 10;
```

Since we can use an int variable in the square braces as the array index, we could rewrite the above as:

```
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49};
double average = 0;
double total = 0;

// variable to use as index
int i = 0;
total = total + grade[i];
i = i + 1; // now the index is 1
total = total + grade[i];
i = i + 1; // 2
total = total + grade[i];
i = i + 1; // 3
total = total + grade[i];
i = i + 1; // 4
total = total + grade[i];
```

```
i = i + 1; // 5
total = total + grade[i];
i = i + 1; // 6
total = total + grade[i];
i = i + 1; // 7
total = total + grade[i];
i = i + 1; // 8
total = total + grade[i];
i = i + 1; // 9
total = total + grade[i];
average = total / 10;
```

If it makes sense to you that the two versions of the code above do the same thing, now we can make the big jump forwards: what we want to do are just the two lines

```
total = total + grade[i];
i = i + 1;
```

over and over. And that's exactly what a for loop is designed to do! We'll make i our for loop counter variable, and update i as part of the for loop. So now we can rewrite the code as

```
int[] grade = {54, 99, 87, 68, 75, 98, 74, 82, 90, 49};
double average = 0;
double total = 0;

// add up all the grades in the array
for (int i = 0; i < 10; i = i + 1) {
    total = total + grade[i];
}

average = total / 10;
```

But what if we have more or fewer grades? Well the initialization of the array would have to change, but the rest of the code wouldn't get any longer. The only part that would have to change is the 10, which is the size of the array. But what if we are getting the grades from a method that doesn't tell us how many there are? We know how to ask the array itself for that value, so we could do

```
int[] grade = getGrades();
double average = 0;
double total = 0;

for (int i = 0; i < grade.length; i++) {
    total = total + grade[i];
}

average = total / grade.length;
```

This will still work even if there are 100 or 1000 or more grades.

When we need to solve a problem involving an array, the answer is almost always to use a for loop, and that for loop almost always has the form

```
for (int i = 0; i < arrayVariable.size; i++){
   do something using arrayVariable[i]
}
```

We would think of **arrayVariable[i] as the "current element of the array in this for loop."**

So let's print the elements of the grades array, but number them. If we are printing for a user to see, remember that we want to print using human numbers, not CS numbers, so we will need to increase the number when we print

```
for (int i = 0; i < grade.size; i++){
   System.out.println("Grade #" + (i + 1) + ": " + grade[i]);
}
```

It's the same for loop, but inside the thing we are doing with "the current element" grade[i] is printing it, with some nice formatting, including adding 1 to the index, so we get something like "Grades #1: 54" etc.

Random Numbers

Suppose I want to have a list of numbers in an array to test some code, but I don't want to spend the time reading in values from a user, and I want to try lots of different values to make sure the code works for any situation. Java provides tools for generating random numbers.

Well… tools for generating *pseudo*-random numbers. Unless we are getting values from some truly random outside source[4], the "random" numbers in computing are always generated by some process that is complex enough to look random, but technically is not actually random. This shouldn't matter unless we are relying on the randomness for privacy or security reasons; in those cases we want to make sure that our results are random enough that some outside person couldn't do the same calculation themselves to find out what our numbers were.

For most purposes, the Random class generates values that are as random as we need. A Random object has a number of behavior methods that can generate random values for Java's primitive types. Since most programs don't need random numbers, you will need to add an import statement to be able to use Random.

First we need to create the Random object, then we can use it to generate actual random values.

```
// needed import at the top of the class for this
// create Random object that generates random values
Random randGen = new Random();

// generate a random int - ANY int value is possible
int anyNum = randGen.nextInt();

// generate a random int from 0 up to but not including 10
int singleDigit = randGen.nextInt(10);

// generate a random double from 0.0 to 1.0 (inclusive)
double d = randGen.nextDouble();

// generate a random true or false (50/50)
boolean coinFlipHeads = randGen.nextBoolean();
```

Notice that there are two versions of nextInt(), the one with no parameters could give any int: large, small, positive, negative, zero. The one that takes an int parameter will always

[4] For instance a company could generate random numbers by pointing a camera at a large number of lava lamps and using their constantly changing shapes to set the binary digits of a random number. Technically, with a complex enough fluid dynamics modeling program and the complete physical specifications of the lamps, the temperature in the room, etc, someone *could* generate these numbers too, but this would be pretty random.

generate an int from zero up to but *not including* the value given, so if we give it 10, we get numbers from 0 to 9.

The nextDouble() method, meanwhile, always generates a double from 0.0 to 1.0. It turns out that this is enough to allow us to get to any range of doubles by doing some basic math[5]. If we want a larger range, we just multiply the result. If we want the range to start higher or lower than zero, we add to or subtract from the result.

```
// generate a random double from 0.0 to 1.0 (inclusive)
double d = randGen.nextDouble();
// generate a random double from 0.0 to 100.0 (inclusive)
double d = randGen.nextDouble() * 100.0;
// generate a random double from -50.0 to 50.0 (inclusive)
double d = randGen.nextDouble() * 101.0 - 50.0;
```

Note that we could do the same kind of adjustment on the result of the version of nextInt that takes a parameter to move its range up or down.

So we can use these random generation methods to fill an array with random values.

Another common way we use random numbers with arrays is to choose a random value from a list. The idea is: first put the list of values you want to choose from into an array, then generate a random number that is a valid index in that array. You can then use the random index to extract a random value from the list:

```
// array of days of the week
String[] days = new String[] { “Monday”, “Tuesday”, “Wednesday”,
     “Thursday”, “Friday”, “Saturday”, “Sunday”};

// generate a random index valid in the array
// a value from 0 up to but not including length of the array
int randIndex = randGen.nextInt(days.length);
```

[5] Doubles in Java represent real numbers in mathematics. There are as many real numbers between 0.0 and 1.0 as there are real numbers total. No, really. This is because the number of real numbers is what we call *uncountably infinite*; this particular class of infinity has this counterintuitive property, which I will happily show you the proof for if you come ask me about it in office hours. It’s a nice proof and doesn’t require any complex math.
To be fair, doubles only *represent* reals; the actual number of doubles is constrained by how many binary patterns we can fit in 64 bits, and so technically doubles work a little differently than reals, but the math still works out well enough in Java for most purposes.

```
// a randomly chosen day from the array
String randomDay = days[randIndex];
```

Sometimes when you are using random numbers to test your code and you discover that you have an error, it is useful to be able to test the code with the same sequence of "random" numbers over and over until you have fixed the problem. There is a parameterized constructor for the Random class that takes a *seed*, a starting value for the "random" calculation used to generate the sequence of random numbers. If you give the constructor a seed, you will always get the same sequence, and then you can change the seed to test with a different sequence, knowing that if you find an error with those values, you'll be able to make changes and test again with the same values

```
// needed import at the top of the class for this
// create Random object that generates random values
// using a seed value so it generates the
// same sequence every time the program runs
int[6] seed = 989;
Random randGen = new Random(seed);

// this will print the same sequence of 100 numbers
// each time the program runs as long as it uses the
// same seed
for (int i = 0; i < 1000; i++) {
     System.out.println(randGen.nextInt());
}
```

Arrays and Methods

Array variables are like any other variables; they can be local variables in any methods, and we can pass them into methods and return them from methods. We just need to use all the rules for arrays and all the rules for methods.

Array parameters

Here a method that takes two arrays as parameters is called by main:

```
// takes an array of ints and an array of booleans
// as parameters
public static void chkArrays(int[] intList, boolean[] booList) {
```

[6] Technically, this should be the type long, another primitive variable type for whole numbers that uses more bits than an int, so we can get more different seeds. We've left long out of our discussions for simplicitly

```
        int iCount = 0; // how many zeroes in int array
        for(int i = 0; i < intList.length; i++) {
            if (intList[i] == 0) {
                iCount++;
            }
        }
        int bCount = 0; // how many zeroes in int array
        for(int i = 0; i < booList.length; i++) {
            if (booList[i] == 0) {
                bCount++;
            }
        }
        System.out.println("There are " + iCount + " zeroes and "
            + bCount + " falses");
}

public static void main(String[] args) {
        int[] numList = new int[]{1, 0, 0, 2, 3, 0, 4, 5, 0};
        boolean[] tfList = new boolean[]{true, false, true,
                            true, false, false, false, true};

        // call the method, passing it the addresses
        // of the two arrays
        chkArrays(numList, tfList);
}
```

Notice that there were no square braces when we passed the arrays into the method. We usually think of this as "passing the whole array" rather than an individual element, but of course what we are really doing is passing the value of the array variable, which is just the memory address of the start of the array on the heap.

Returning Arrays

Now let's have a method that returns an array.

```
// method that returns an array of ints
public static int[] getGradeList() {
        Scanner scan = new Scanner(System.in);
        System.out.println("How many grades");
        int howMany = scan.nextInt();
```

```
        // create array to return later
        int[] gradeList = new int[howMany];

        // read in grades
        for (int i = 0; i < gradeList.length; i++) {
            System.out.println("what is grade #" + (i+1));
            gradeList[i] = scan.nextInt();
        }

        // return the memory address of the array on the heap
        return gradeList;
}

public static void main(String[] args) {
        int[] grades = getGradeList();
        // code to do things with grades…
}
```

Again, when we return, we don't use square braces because we are returning "the whole array" or rather, the address on the heap where the array was created. As usual, if a method returns an array, it is important that we catch that result in a variable or pass it to another method (we can't just stick it in println, because arrays print as memory addresses, not the contents).

Arrays in Classes

Array variables are like any other variables; they can be used as instance variables if the design of a class includes a list of values of some type.

Here is a the start of a class with a boolean array as an instance variable. We are currently marking it public, but only so the example can more clearly show how array instance vars and classes interact, it *should* be private like any other instance variable.

Like any other instance variable, an array instance variable starts with a default value, null, and it is the default constructor's job to set up array instance variables, in a way appropriate to how the class uses them. Some array instance variables might be left null until they are needed during a behavior method, others we may know the length of in the constructor, and we might even know some values.

In this class representing a vacation cabin, we have an array for keeping track of which months the cabin is booked, and so in the constructor we set up the array to be length 12. We'll also decide that by default the owners always use the cabin for themselves in December:

```
public class Cabin {
     // which months is the cabin booked
     // currently null
     public boolean[] monthsBooked;
     // how many bedrooms it has
     private int bedrooms; //accessor, mutator…

     // default constructor sets up the instance vars
     public Cabin() {
          setBedrooms(1);
          // twelve months in the year
          monthsBooked = new boolean[12];
          // always booked in December
          monthsBooked[11] = true;
     }
}

// in a main…
Cabin summerPlace = new Cabin();
// booked in June, July, August
     // "summerPlaces' monthBooked array sub 5 is true"
summerPlace.monthsBooked[5] = true;
summerPlace.monthsBooked[6] = true;
summerPlace.monthsBooked[7] = true;
```

Because we made the array instance variable public, we could use the dot operator to access it from the Cabin variable, and then use the square braces to go inside the array and set some of its elements. In memory:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
summerPlace	51	MM100
	…	
[heap]		
	100	monthsBooked ~~null~~

		MM200 bedrooms ~~0~~1
	...	
Index: 0	200	false
1	201	false
2	202	false
3	203	false
4	204	false
5	205	true
6	206	true
7	207	true
...	...	...
11	211	true
length	212	12
	...	

Accessors and Mutators for Array Instance Variables?

We know that making the array instance variable public was wrong, because instance variables should never be public. For all other instance variables, we have created accessors and mutators. We could do this for array instance variables:

```
public class Cabin {
	// which months is the cabin booked
	// currently null
	public boolean[] monthsBooked;
	// how many bedrooms it has
	private int bedrooms; //accessor, mutator…

	public boolean[] getMonthsBooked() {
		return monthsBooked;
	}

	public void setMonthsBooked(boolean[] newMonths) {
		if (newMonths != null && newMonths.length == 12) {
			monthsBooked = newMonths;
		}
	}
}

// …in a main
```

```
Cabin winterPlace = new Cabin();
winterPlace.getMonthsBooked()[0] = true; // January
winterPlace.getMonthsBooked()[1] = true; // February

Cabin inTheWoods = new Cabin();
boolean[] octOnly = new boolean[12]; // all false
octOnly[9] = true; // booked in spookiest month
inTheWoods.setMonthsBooked(octOnly);
```

Notice that since getMonthsBooked() returns an array, we were able to use square braces directly on its result, combining the rules for a method that returns (an array) with the rules for arrays.

But actually, we would not usually provide public accessors and mutators for an array. Once an outside class like main has access to the memory address of our array instance variable, it has complete control over the contents of that array, which violates our principle of encapsulation; there's no point in making the array private if the accessor and mutator actually give outside classes complete control over the array anyway.

In some cases, we might create an accessor an mutator that always make a copy – the accessor copies the instance variable and returns that copy, the mutator makes a copy of the parameter array and stores the copy in the instance variable. This way we get the accessor and mutator behavior we want but the outside class can't mess up our array.

But it is far more common just not to have the accessor or mutator for an array instance variable at all, and instead provide other methods that allow outside classes to try to affect or view the contents of the array without exposing the address of the whole array. For the purposes of this course, arrays are the exception to our usual rule: you should not create accessors or mutators for array instance variables unless the assignment says so explicitly.

Searching Arrays

Arrays give us a way to save large lists of data. Sometimes we want to just process every element, but sometimes we want to know if an array contains a specific value, and if so, where in the array (at what index) that value occurs. For simplicity we will assume that either the value only appears once or that we are always interested in the first copy of a value.

Just Finding if a Value Is In The Array

Suppose that we want to know if there is a 0 in our array of grades. If we just need to report whether or not it is there at all, we could say

```
// mysterious method returns array with values
int[] grade = getGradeList();
// flag for whether we found a zero
boolean found = false;

for (int i = 0; i < grade.length; i++) {
    // check whether the current value is 0
    // if so, update the flag
    if (grade[i] == 0) {
        found = true;
    } // no else here!
}
// once we are done checking the whole array, report the result
if (found) {
    System.out.println("found a zero");
} else {
    System.out.println("no zeroes found");
}
```

We used a boolean called found to record whether or not we found a zero. Using a for loop as usual, we checked each element of the array. If the current element was a zero, we updated the found variable.

Notice that there is no else on the if inside the for loop. If the current element grade[i] *is* zero, then we want to update our boolean to true. But if it isn't, does that mean we should do something else? Update the boolean to false? NO! If the current element isn't zero, that just means we haven't found a zero *so far*. We still have the rest of the array to search.

In fact, adding an else that sets the boolean to false would be a disaster! If we haven't found our zero yet, then the boolean is still false, so setting it to false would be a waste of time. But what if we found the zero, set found to true, and then looked at the next number, which isn't zero… we'd be setting it back from true to false and losing our information!

The only time we can reasonably have an else, to report whether it was there or not, is after we have checked the entire array – after the for loop.

Note that, if we really are only interested in the first occurrence, or know that there is only one, then we don't actually need to keep searching once it is found. So we could adjust the loop condition so that we stop once it is found.

```
// mysterious method returns array full of grades
int[] grade = getGradeList();
// flag for whether we found a zero
boolean found = false;

// goes through whole array but stops if found
for (int i = 0; i < grade.length && !found; i++) {
    // check whether the current value is 0
    // if so, update the flag
    if (grade[i] == 0) {
        found = true;
    } // no else here!
}
// once we are done checking the whole array, report the result
if (found) {
    System.out.println("found a zero");
} else {
    System.out.println("no zeroes found");
}
```

Finding the Location of a Value

Suppose we want to know *where* in the array a value occurs.

```
int[] grade = getGradeList();

// let the user choose a value to search for
System.out.println("What value to find?");
int seeking = scan.nextInt();

for (int i = 0; i < grade.length; i++) {
    if (grade[i] == seeking) {
        // i is the index in the array where we
        // found the value, report it if found
        System.out.println("found a " + seeking +
" at position " + i);
```

```
    }
}
```

In this case instead of just searching for zero, we made our code more general by getting a value, seeking, to search for from the user. Our loop looks much the same, but this time we said not just that we had found it, but the position in the array where we found it:, i. Remember that i is a computer science number; if we are really going to print to a user, we might want to print (i+1) instead. But actually, when searching for a location, we often want to store that in a variable to use later in the program:

```
int[] grade = getGradeList();

System.out.println("What value to find?");
int seeking = scan.nextInt();

int foundAt = -1; // index position in the array

boolean found = false; // whether we found it

for (int i = 0; i < grade.length && !found; i++) {
    if (grade[i] == seeking) {
        foundAt = i; // store the position
        found = true; // store that we found it
    }
}

if (found) {
    System.out.println("found " + grade[foundAt] + " at " +
     foundAt);
} else {
    System.out.println(seeking + " not found");
}
```

In this version, we used the foundAt variable to store the index where we found the seeking value, so that it would be available after the for loop. We also used the found variable so that we would know whether we found it or not.

Note that we started foundAt as -1. We'll see why this is a good idea in a moment.

Finding Location and Recording Success in One Variable

It is much more common, instead of using both an int for position and a boolean, to just use the int. We can simply set the int to an impossible value for an index to start with, and if it still has that value after the loop ends, we know we must never have found the value. The standard value to indicate something was not found is -1, since that is not a valid index for any array:

```
int[] grade = getGradeList();

System.out.println("What value to find?");
int seeking = scan.nextInt();

int foundAt = -1; // position starts with impossible value
// for loop goes through whole array
// but stops early if foundAt ever changes
for (int i = 0; i < grade.length && foundAt == -1; i++) {
    if (grade[i] == seeking) {
        foundAt = i;
    }
}
// after loop, if foundAt is still -1 it was never changed
// so we never found it; if we did change it, it was found
if (foundAt != -1) {
    System.out.println("found " + grade[foundAt] + " at " +
     foundAt);
} else {
    System.out.println(seeking + " not found");
}
```

We started foundAt as -1. If we found the value we were seeking, we changed it to the current value of i. We can then use whether foundAt is still -1 both to help us end the for loop early and to check after the loop whether we found it.

Parallel Arrays

Each array can only store one type of data; we can have an array of ints and separately an array of strings, but one array can't store both ints and strings. If we have multiple lists of different types that are associated with each other, we can set them up as *parallel arrays*: multiple arrays of the same length where we use the same index across all arrays to group data.

Suppose that in addition to the array of grades, we have arrays of the first and last names of the students with those grades. Since there are the same number of first names and last names and grades, the arrays will be the same size, and we can use the same index to indicate the same person, grouping their first name, last name, and grade. So if we say:

```
int[] grade = new int[10]; // grades of students
String[] fNames = new String[10]; // first names of students
String[] lNames = new String[10]; // last names of students
fNames[0] = "Jan"; // student #0's first name
lNames[0] = "Smith"; // student #0's last name
grade[0] = 85; // student #0's grade

// all info about student #1, in three arrays
fNames[1] = "Gan";
lNames[1] = "Djones";
grade[1] = 90;

// Stan Melchizadek got a 99
fNames[2] = "Stan";
lNames[2] = "Melchizadek";
grade[2] = 99;
```

We are saying that there is a student whose name is Jan Smith whose grade is 85 – since all those values are at position 0 in their respective arrays, they go together. Similarly, Gan Djones got a 90 and Stan Melchizadek got a 99. You could think of the parallel arrays as each being a column in a table, and each index indicating a row in that table with all the information about a student.

Since parallel arrays must be the same size, we can use a single for loop to go through them all at the same time:

```
// use special initialization to create parallel arrays
int[] grade = {85, 90, 99, 80, 73};
String[] fNames = {"Jan", "Gan", "Stan", "Nan", "Ann"};
String[] lNames = {"Smith", "Djones", "Melchizadek", "Brown",
     "Whitley"};

// for loop can go through all three arrays at once
```

```
// since they are lined up and all the same size
for (int i = 0; i < grade.length; i = i + 1) {
    System.out.println(fNames[i] + " " + lNames[i] + " has " +
     grade[i]);
}
```

If we search one array to find the position of a value, we can use the index we found to look up the related information in the parallel arrays. For instance, if we look up the last name of a student we can find the matching first name and grade.

```
// assuming we set up the three arrays already

System.out.println("Look up what surname?");
String lookFor = scan.nextLine();

int foundAt = -1;
for (int i = 0; i < lNames.length; i++) {
    // finding the location of the last name
    if (lNames[i].equals(lookFor)) {
        foundAt = i;
    }
}
// whichever position we found the last name,
// the matching first name and grade will be at
// the same position, in the other arrays
if (foundAt != -1) {
    System.out.println(fNames[foundAt] + " " + lNames[foundAt] +
     " has grade " + grade[foundAt]);
} else {
    System.out.println("No such student");
}
```

Parallel arrays do rely on carefully maintaining the size and locations of data. If one of our arrays has data in the wrong order, everything breaks. Creating a class that has instance variables for each type and then having a single array of such objects will usually be better than relying on parallel arrays. Parallel arrays can be more efficient in some situations, however, and are great for practicing array code.

Arrays with FirstEmpty

Arrays are a static sized data structure, which means that we have to choose the size of the array when we create it, and we cannot change the size later, we can only make a new array of a different size, and copy the data into the new one. (Don't forget, like variables based on classes, array variables are references, they only hold the *address* of the actual array object on the heap, so we could keep the same array variable and set it to the memory address of a larger array, but that isn't the same as changing the size of the existing array object.)

For this reason, we often create an array larger than the data we start with and then add values to it as the program runs; if we don't know how much data we will be dealing with, we will usually err on the side of making the array too large, because wasting some space is better than wasting a lot of time copying elements from one array to another.

Suppose we have an array that is already partially full, and a new value has just come in. We want to store the new value in the array, but not at a position where we have already stored a previous value.

One option would be to loop through the array until we find an empty spot… but there are several reasons this is a bad idea.

First: how do we know whether a spot is empty? If the array started with default values such as 0, false, or null, we could check for those, but false is certainly a valid possible value for a boolean array element, 0 is often valid for a number, and null could well be valid for any other type. We are back to the problem we had with sentinel values: this only works if there is some value that is not valid data.

But second: even if we do have some value we can use to indicate empty spots, looping through the array to find the first such spot every time we want to add is an *enormous waste of time*.

When we measure the time for a program to do a task, we measure not in terms of seconds elapsed, but based on how the number of steps is related to how much data there is: Suppose we are working with 10 data values in our array. Each time we search the array we have to look at on average (1/2)*10 items, and we have to do this 10 times, so (1/2)*1000*1000. If it were 1000 values, it would be (1/2)*1000*1000. So in general, if we are faced with n values, we have to do (1/2)*n^2.

We usually ignore the constant coefficient at the front: whether we are actually doing 18 lines of code to process each value, or just 3 lines, that number *(1/2) gets dwarfed by the number of values. So we ignore the (1/2) and just simplify this down to saying: doing it this way costs n^2 steps. That's not great if n is 10. That's catastrophic if n is 1000.

Especially when I tell you that we can get the same result in just n steps, if we use just one extra int-worth of space!

We will keep an int variable like a sort of bookmark for the first spot in the array that is not currently occupied: the *first empty index*. We often call this variable something like firstEmpty or firstEmptyIndex or fei. Since array indices are always ints, it is always an int.

Suppose we are getting names from the user, and filling an array using a firstEmpty.

```
// the array so far, partially full
String[] nameList = new String[100];
nameList[0] = “Abe”;
nameList[1] = “Bea”;
nameList[3] = “Cam”;
nameList[4] = “Deb”;
nameList[5] = “Eve”;
// the firstEmpty so far
int firstEmpty = 6;

final String QUIT = "QUIT";
System.out.println("what name? " + QUIT + " to quit");
String username = scan.nextLine();

// while the user doesn’t quit
// and there is still space in the array
while (!username.equals(QUIT) && firstEmpty < nameList.length) {
    // put the new name at the current first empty spot
    nameList[firstEmpty] = username;
    // update firstEmpty so next name goes at next spot
    firstEmpty = firstEmpty + 1;
    // get nextname from the user
    System.out.println("what name? " + QUIT + " to quit");
    username = scan.nextLine();
}

// if we ran out of space,
// let the user know that’s why we quit
if (firstEmpty >= nameList.length) {
    System.out.println("ran out of space");
}
```

The heart of this code is just putting the user name at the firstEmpty index in the array, and then incrementing the firstEmpty index, since that spot in the array isn't empty anymore.

In this case, we wanted to read multiple names, so we did use a while loop, but notice this is one of those rare cases when solving a problem with an array does not need a for loop!

If we had been searching for the first empty spot, instead of using an int to keep track of it, doing that would have needed a for loop nested inside the while, which would have taken the very slow n^2 time.

Notice that we did check each time whether the firstEmpty had fallen off the end of the array by checking it against the array's size. If we run out of space, then we either have to give up, or create a new, larger array, copy all the elements from the old array into the new, and continue from there (this is slow! So staring with an oversized array is smart.).

We put these values into the array because we have some use for them. If we need to process the values in our array we need to go through all of them, and that brings us back to using a for loop, but when we have an array with a firstEmpty, we don't usually want to process the whole array because there are probably a bunch of empty spots at the end of the array that we don't want to include in whatever process we are doing. So, the for loop for dealing with an array that has a firstEmpty has a condition that compares the counter to the firstEmpty, not to the size of the array:

```
// print the students in the array
// but not the empty spots at the end
for (int i = 0; i < firstEmpty; i++) {
    System.out.println("Student #" + (i + 1) + ". " +
     nameList[i]);
}
```

Copying and Resizing Arrays

To copy an array, we must make a new array of the same size, and copy each element

```
// mystery method creates the original array
String[] original = getOriginalArray();

// the new array to copy into
// same size as original
String[] destination = new String[original.length];

// these arrays function like parallel arrays
```

```
// so we can use one loop to go through them
for (int i = 0; i < original.length; i++) {
    destination[i] = original[i];
}
```

If we planned poorly and ran out of room in our array, we can't make the array bigger so we need to make a new, larger array and copy all the elements over. One good guideline is to make the new array at least twice the size of the old.

```
// the original array
String[] original = getOriginalArray();
// the new array to copy into
// twice the size of original
String[] destination = new String[2 * original.length];

// still just one loop, but we must use
// the smaller size, we are not filling up the
// larger array, just leaving the rest of it empty
for (int i = 0; i < original.length; i++) {
    destination[i] = original[i];
}

// if we did this because we ran out of space, but need to
// continue working with the "same" array, we might as well
// move the new array's address into that old variable
original = destination;
```

Notice that if we were doing this for an array with a firstEmpty, the firstEmpty doesn't need to change. If it reached the size of the original array, that's okay, because that's now a valid index in the new bigger destination array and we can go on adding to it.

Object Arrays

As usual in Java, we can combine any tools we have, and we just need to remember to apply all the rules of both. We can have an array of any type valid for variables, and we can use classes to create types for variables, so we can have an array of any type we have made a class for.

When we create an array of a class type, as always all the elements start as the default value for variables of that type, null. So we will need to use new more than once: we use

new with square braces to set up the array, and we also use new with parentheses to create individual instances of the class whose memory addresses are stored in the array.

Let's assume we have a Dog class with instance variables for name and number of sticks, and the ability to bark. For the moment, all this will be public in Dog (we know better than this for instance variables, but it will help simplify the initial example).

```
1   // array of dogs variable - no array, no dogs yet
2   Dog[] dogList;
3   // create array, all elements null, no dogs yet
4   dogList = new Dog[10];
5   // create an actual dog, address located in the array
6   dogList[0] = new Dog();
7   // use the dog variable in the array
8   dogList[0].name = "Spot";
9   dogList[0].sticksFetched = 3;
10  dogList[0].bark();
```

In line 2 we have declared the variable dogList, whose type is array of Dogs, but we have used new 0 times, so there is not even an array yet, and certainly no Dog objects. At that point memory looks like

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
dogList	51	*no value*
	...	

In line 4 we use new with square braces, and this creates the array object on the heap, which is a list of Dog *variables* each of which could hold the address of an actual Dog object, but they are all null at this point. Now memory looks like:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
dogList	51	MM100
	...	
[heap]		
Index: 0	100	null
1	101	null
2	102	null
3	103	null

4	104	null
5	105	null
6	106	null
7	107	null
8	108	null
9	109	null
length	110	10
	...	

In line 6 we use new with parens to create an actual Dog object, and put the memory address of that Dog object into the 0th row in the array. Now memory looks like:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
dogList	51	MM100
	...	
[heap]		
Index: 0	100	MM200
1	101	null
2	102	null
3	103	null
4	104	null
5	105	null
6	106	null
7	107	null
8	108	null
9	109	null
length	110	10
	...	
	200	name: null sticksFetched: 0 ...
	...	

The rest of the code is affecting that Dog object, using the variable name dogList[0]. Again, this may feel like a weird variable name at first, but it is just a variable, so we can use the dot operator on it like any other Dog variables. By the end of the code, memory looks like:

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
dogList	51	MM100
	...	

[heap]

Index: 0	100	MM200
1	101	null
2	102	null
3	103	null
4	104	null
5	105	null
6	106	null
7	107	null
8	108	null
9	109	null
length	110	10
	...	
	200	name: “Fluffy” sticksFetched: 3 ...
	...	

Object Array Special Initialization

Since object arrays are arrays, we can use special initialization on them just like other arrays. We could do this by first creating the objects in individual variables and then putting them into the array, but it would be more standard just to create the objects in the special initialization.

```
// assuming a parameterized constructor in Dog
// create individual Dogs (probably a waste of time
// unless we will also need these variables for individual
// use)
Dog d1 = new Dog(“Muffy”, 1); // Dog has param constructor
Dog d2 = new Dog(“Fluffy”, 20);
Dog d3 = new Dog(“Clyde”, 300);
// put the addresses from the individual variables
// into the array
Dog[] list1 = new Dog[]{d1, d2, d3};

// more normal way to do it: just create the Dog objects
// as part of special initialization
Dog[] list2 = new Dog[] {     new Dog (“Spot”, 44),
                              new Dog(“Rover”, 55),
                              new Dog(“Mr. Dog”, 66) };
```

Object Array Instance Variables

Since object arrays are variables, they can be instance variables inside other objects. Here is a class that has instance variables for arrays of objects:

```
public class Zoo {
     private String zooName;
     public Penguin[] penguins;
     public Lion[] lions;
     public Zebra[] zebras;

     public Zoo() {
          setZooName("Generic Zoo");
          // we have space for 10 penguins
          penguins = new Penguin[10];
          // we start with one penguin
          penguins[0] = new Penguin("Willie");

          // we have space for 5 lions
          lions = new Lion[5];
          // but we didn't get any lions yet

          // we don't yet have space for any zebras by default
          // but we might set up the zebra array later
     }
}
```

In this class representing a zoo, we have three instance variables arrays of objects based on classes representing penguins, lions, and zebras. The default constructor sets up the array of penguins and adds one actual penguin, sets up the space for lions, but does not set up the space for zebras by default. We know better, but I made the variables public to make it easier to see what's going on in this example.

In main we could now say

```
Zoo tinyZoo = new Zoo();
```

As a result of this line, the default constructor runs, setting up a lot on the heap!

variable	*MM*	*value*
[stackframe: main (args)]		
args	50	null
tinyZoo	51	MM100

	...	
[heap]		
	MM100	zooName ~~null~~ "Generic Zoo" penguins ~~null~~ MM200 lions ~~null~~ MM400 zebras null
	...	
Index: 0	200	MM300
1	201	null
2	202	null
3	303	null
	...	
9	105	null
length	110	10
	...	
	300	penguinName ~~null~~ "Willie" ...
	...	
Index: 0	400	null
1	401	null
2	401	null
3	403	null
4	404	null
length	405	5

Because we made the instance vars public, we could say things like this in main:

```
Zoo tinyZoo = new Zoo();
tinyZoo.setName("The Tiny Zoo");
// give them a lion
tinyZoo.lions[0] = new Lion("Simba");
// set up space for and give the zoo three Zebras
tinyZoo.zebras = new Zebra[] { new Zebra(), new Zebra(),
     newZebra()};
```

Make sure you understand why tinyZoo.lions[0] is valid but tinyZoo[0] is not. The tinyZoo variable *contains* an array, but it *is not* an array itself.

Arrays of Objects that Contain Arrays

We can just continue combining components of Java: we can have an array of any type, and a class containing an array of something is a valid type, *so* we can create an array of objects each of which contains an array of objects (each of which could contain an array of objects, etc, etc.

Continuing with the Zoo class, we could say in main:

```
// a single tiny zoo
// assume parameterized constructor that takes name
// and chains to default
Zoo tinyZoo = new Zoo("The Tiny Zoo");

// an array that can hold the addresses of multiple zoos
Zoo[] zooFederation = new Zoo[10];

// put the tiny zoo's address into the array of zoos
zooFederation[0] = tinyZoo;
// add another zoo to the federation
zooFederation[1] = new Zoo("Baltimore Zoo"):
zooFederation[2] = new Zoo("National Zoo");
zooFederation[3] = new Zoo("Salisbury Zoo");
// … code for 96 other zoos

// … code that sets up their animals

// let's look at all the lions
for (int i = 0; i < zooFederation.length; i++) {
  if (zooFederation[i].lions != null) {
     for (int j = 0; i < zooFederation[i].lions.length; j++);
       if (zooFederation[i].lions[j] != null) {
          // print the lion
          System.out.println(zooFederation[i].lions[j]);
          // have the lion roar
          zooFederation[i].lions[j].roar();

       }
     }
  }
```

```
}
```

So we have an array of zoos, each of which contains an array of Lions (which may or may not be null, and may or may not contain addresses of actual lion objects). We have nested for loops, one to go through the array of zoos, and one that for each zoo goes through the array of lions in that zoo (checking for nulls along the way).

Make sure you understand why zooFederation[i].lions[j].roar() is valid but zooFederation[i].lions.roar() or zooFederation[i]. [j].roar() or zooFederation.roar() are not. The zooFederation variable stores the address of an array of zoos, each element of which has an instance variable that stores the address of an array of lions, each element of which stores the address of a lion that can roar. The array of lions isn't a lion that can roar, and neither is the array of zoos, and we can't put the dot operator between subscripts, that's just a syntax error[7].

[7] Two square braces side by side *can* be valid syntax. Since arrays are a valid type for variables, and we can have an array of any valid type, we can have an array of arrays – an array whose elements hold the memory addresses of other arrays. We will cover these multidimensional arrays in Java II.