Chapter 5

Arrays and Searching

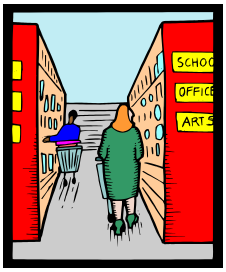
What is in this Chapter ?

When solving problems, we often deal with data that has been collected together. We often must sift through collections of information to find answers. This chapter discusses how data can be collected together into ***Arrays*** and also the various ways that we can ***search*** through the data efficiently to find what we want.

5.1 Storing Objects Together Using Arrays

In real life, objects often appear in groups or collections. For example, we see collections of objects when we are...

- storing products on shelves in a store
- maintaining information on multiple customers
- keeping track of cars for sale, for rental or for servicing
- a personal collection of books, CDs, DVDs, cards, etc...
- maintaining a shopping cart of items to be purchased from a website





As we have already learned, data is stored in variables. As this point, we have seen variables that are one of 8 kinds of primitive data types: (**byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, **char**). Each variable stored one value.

```
int         age = 18;
char        gender = 'f';
double      weight = 145.2;
boolean     retired = true;
```

Sometimes, however, we need **many** variables to solve our problems. For example, imagine writing a program that asks the user to enter the ages and genders of a large group of people (e.g., 100) and then trying to find the oldest, youngest, average age, number of adults, number of pre-teen girls, number of teenage boys, number of retired people, ... maybe even grouping them into categories of minors, adults and post-retirement etc...

We may begin by getting the individual ages and genders from the user as follows:

```
int      a1, a2, a3, a4, a5,  ... etc..., a100;
char     g1, g2, g3, g4, g5,  ... etc..., g100;

Scanner     keyboard = new Scanner(System.in);

System.out.println("Enter age 1:");
a1 = keyboard.nextInt();
System.out.println("Enter gender 1:");
g1 = keyboard.next().charAt(0);

System.out.println("Enter age 2:");
a2 = keyboard.nextInt();
```

```
System.out.println("Enter gender 2:");
g2 = keyboard.next().charAt(0);

System.out.println("Enter age 3: ");
a3 = keyboard.nextInt();
System.out.println("Enter gender 3:");
g3 = keyboard.next().charAt(0);

... etc...

System.out.println("Enter age 100: ");
a100 = keyboard.nextInt();
System.out.println("Enter gender 100:");
g100 = keyboard.next().charAt(0);
```

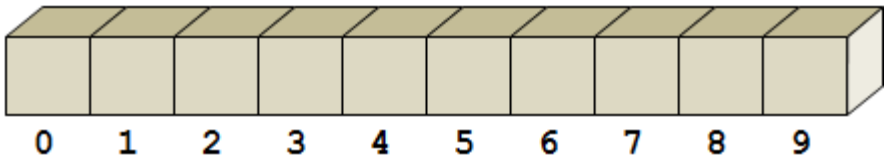
Can you imagine how long this code would be and how much code duplication is involved ? What if there were 1000 people ? Yikes!

The problem is that each age and gender is stored in their own unique variables for later processing to determine the various requested statistics and to group them into categories of minors, adults, post-retirement, etc...

Rather than having to create 200 variables ... there is a better way to do this. Most (if not all) programming languages have a fixed data type called an ***array***.

*An **array** is a collection of data items that can be selected by indices (or locations).*

So, arrays are a means of "gluing" a bunch of variables side-by-side in some specified order.



Each ***item*** (also known as ***element***) in an array is stored at a location which is specified by an ***index***. The **index** is an integer that identifies the location within the sequence of items. Indices start at 0.

Arrays can hold a fixed number of items … that is … they don't grow or shrink … they have a fixed size. In the above image, the array holds exactly 10 items, so the indices go from **0** through **9**. An index of **10** or higher would be out of the array's "bounds", and would therefore be invalid.

We often refer to the "size" of the array as its ***length***. The size/or length of the array is NOT the number of items that we have put inside it, rather, it is the ***capacity*** of the array (i.e., the maximum number of items that we can put into it).

Arrays are ***data types*** that store a particular kind of item within them. Each item is understood to be of the same type (e.g., all integers, all floats, all booleans, etc..).

Arrays are declared similarly to regular variables but square brackets are used to tell the compiler that the variable will hold many values instead of just one:

Single-value variables	Array variables
`boolean hungry;`	`boolean[] hungryArray;`
`int days;`	`int[] daysArray;`
`char gender;`	`char[] genderArray;`
`float amount;`	`float[] amountArray;`
`double weight;`	`double[] weightArray;`

(* Note that the square brackets may appear either with the type (as shown above) or with the variable's name as follows: `int days[];`)

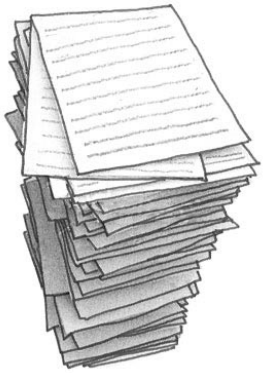
Do you understand the difference between `int days;` and `int[] daysArray;` ? **days** stores a single integer, while **daysArray** stores many integers.

The `int[] daysArray;` variable declaration simply "reserves space" to store the array, but it actually does not create the array object itself.

So, the following code would print out **null** because the variable is not yet initialized:

```
int[]   daysArray;
System.out.println(daysArray);
```

Notice above that we do **not** use the square brackets **[ ]** when you are ***using*** the array variable in our code ... we only use the brackets when we ***define*** the variable.



To ***use*** this variable (which is just like any other variable), we must give it a value. What kind of value should it have ? It should be an **array object**. The array object will hold many integers inside of it, but the variable itself is considered to be a single object. An array is kind of like a "stack of papers". There are many papers, but altogether they make up a single stack.

If we know the values that we want to put into the array, we can simply supply these values on the same line as the declaration of the variable. Here are some examples:

```
int[]       ages = {34, 12, 45};
double[]    weights = {4.5, 23.6, 84.1, 78.2, 61.5};
boolean[]   retired = {true, false, false, true};
char[]      vowels = {'a', 'e', 'i', 'o', 'u'};
```

Here, the array's size is automatically determined by the number of values that you specify within the braces, each value separated by a comma.

At any time, if we would like to ask an array how big it is (i.e., its size or capacity), we can access one of its special attributes called **length** as follows:

```
System.out.println(ages.length);         // displays 3
System.out.println(weights.length);      // displays 5
System.out.println(retired.length);      // displays 4
System.out.println(vowels.length);       // displays 5
```

Notice the use of the dot after the array name, followed by the word **length**. Remember that the **length** of an array is its overall capacity, it is NOT the number of elements that you put into the array.

So then, how do we access the items that are in an array ? Well, each item in the array has it's own unique integer *index*, representing it location. So we need to ask the array for the item that we want by supplying its index. We do this by specifying the array name, followed by square brackets with the index inside:

```
double[]    weights = {4.5, 23.6, 84.1, 78.2, 61.5};

System.out.println("The first weight is: " + weights[0]);
System.out.println("The last weight is: " + weights[4]);
System.out.println("The last weight is: " + weights[weights.length - 1]);
```

Notice that the first item in the array is accessed at position 0. The last item in this particular array is at position 4 ... or the array's length minus 1. The above code would print out the numbers **4.5**, **61.5** and **61.5**.

A common task with an array is to *loop* (or *iterate*) through it. For example, we can sum up all of the weights by using a **for** loop as follows:

```
double sum = 0;
for (int i=0; i<weights.length; i++)
      sum = sum + weights[i];
```

Notice that the loop goes from position 0 in the array to position (length - 1).

A common error when looping through arrays is to go too far past the bounds of the array. For example, consider the following similar code:

```
double sum = 0;
for (int i=0; i<=weights.length; i++)
      sum = sum + weights[i];
```

We have changed the < symbol to <=. This will allow the loop to go one extra time, so the value of loop variable **i** will go from 0 to 5. On the last iteration through the loop, the code will attempt to access the element at position **5** in the array. This will generate an error ... since the array only has positions from **0** through **4**. Java would halt the program and give us an error like this :

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
```

This is telling us that the **main()** method has an error and that the error is of the type: "**Array Index Out of Bounds**". The value of **5** is even shown to indicate the index that we are trying to access within the array.

In addition to being able to access the values in an array, we can modify them.

While we cannot "*add*" extra values to an array or "*remove*" any, we are able to "*replace*" the value at any position in the array with a new value at any time. We do this just as we would modify any variable ... but we must always remember to supply the index position of the element that we want to modify:

```
int[]   ages = {34, 12, 45};

System.out.println(ages[1]);  // displays 12
ages[1] = 13;
System.out.println(ages[1]);  // displays 13 now
ages[1]++;
System.out.println(ages[1]);  // displays 14 now
ages[3] = 8;                  // generates an ArrayIndexOutOfBoundsException
```

A common problem when using an array is to forget to initialize it ... that is ... we might forget to create the array itself:

```
int[]   daysArray;

System.out.println(daysArray[1]); // generates a compile error
```

The above code will not compile. It will give this error:

```
error: variable daysArray might not have been initialized
```

Java is telling us here that we forgot to create the array. This would fix it:

```
int[]   daysArray = {31, 28, 31};
System.out.println(daysArray[1]);
```

Sometimes, however, we are unsure as to what values we want to put into an array. For example, what if we want to enter peoples ages and genders, as in the program at the beginning of this chapter ? We initially may know that there is a maximum number of people (e.g., 100), but we don't know the information yet. Java allows you to create a kind of "blank" array that we could insert items into later. This is like buying a shelf and setting it up, but at first it remains empty until we start loading it up with things.

The syntax that we use in Java to create a kind of "blank" array is as follows:

```
new ArrayType[ArraySize]
```

This is the template to create an array that can hold up to ***ArraySize*** values of type ***ArrayType***. Remember that arrays are fixed size, so you cannot enlarge them later. If you are unsure how many items that you want to put into the array, you should chose an over-estimate (i.e., a maximum bound) for its size. Here are some examples of how to give a value to array variables by creating some fixed-size arrays:

```
int[]     counts = new int[30];      // creates array to hold 30 ints
float[]   weights = new float[100];  // creates array to hold 100 floats
```

The **counts** array will reserve space for 30 integers, while the **weights** array will reserve space for 100 floats. We can do this now:

```
int[]   daysArray = new int[3];

System.out.println(daysArray[1]);
```

The code will compile now, however, the value printed will be zero. That is because whenever we create an array like this, we are simply reserving space, but are not assigning any values for the items in the array. All arrays that store numbers will have **zeroes** as the default values at each position in the array. Boolean arrays will have **false** values as default. Other than that, arrays created in this manner work the same way as before:

```
int[]   daysArray = new int[3];

daysArray[0] = 31;
daysArray[1] = 28;
daysArray[2] = 30;
System.out.println(daysArray[1]); // prints 28
```

5.2 Searching Arrays

In real life, we are often faced with the problem of sifting through information (i.e., data) to determine whether or not a particular type of value lies within the information. We call this the ***searching*** problem, since we are searching through data for a (possibly partial) solution to our problem at hand.

For example, we may look through a list to:

1. determine the existence of a piece of data (e.g., check if a person is on a list)
2. verify that all items satisfy a condition (e.g., have all employees signed up for a seminar)
3. pick an appropriate candidate(s) for our problem (e.g., find an available seat on a flight) etc..

A ***linear search*** (or ***sequential search***) is a method of checking every one of the elements in a list, one at a time and in sequence, until a value is found that satisfies some particular condition (e.g., until a *match* is found).

When developing algorithms, the simplest approach is often called a "**brute force**" approach, implying that there is a lack of human sensibility in the solution. A "brute force" algorithm is one that is often easy to come up with, but that does not usually consider efficiency or any form of ingenuity.

A linear search is the simplest kind of search since it involves naively looking through the elements in turn for the one that matches the criteria. In all the examples below, we will assume that the items that we are searching through are stored in arrays.

Example: (searching a list for a particular item)

Suppose that we had a stack of a few trading cards and that we wanted to know if we had a specific card number (i.e., each card has a unique number on the back). Write a program that goes through an array of these card numbers and determines whether or not a particular card is in the set. We will "assume" that the card numbers have already been read into our program and have been placed into a fixed array and we will ask the user for a specific card number to look for. To begin, here is what we will start with:

```
import java.util.Scanner;

public class CardSearchProgram {
	public static void main(String[] args) {
		boolean found = false;

		int[]  cards = {2, 4, 9, 11, 12, 43, 45, 65, 76, 13, 84, 92, 95, 104};

		System.out.println("Which card are you looking for ?");
		int num = new Scanner(System.in).nextInt();

		// ... complete the code ...

		if (found)
			System.out.println("Card #" + num + " is in the set.");
		else
			System.out.println("Card #" + num + " is not in the set.");
	}
}
```

Notice that we begin with **found** set to **false**, meaning that we assume that the card is not there. To determine if a card is in the set, we then simply check all of the cards and if we find it, we set the boolean **found** to **true**.

This boolean variable is called a ***boolean flag*** because it is like holding up a "sign" (or flag) indicating that something important has just happened. It is analogous to the little flag on mailboxes that the mailperson lifts up to let people know when their mail has arrived. Flags are usually set once during an algorithm and then after the flag is "checked for" in the code, it is sometimes reset for the next round or iteration. Here is the missing code:

```
	for (int i=0; i<cards.length; i++) {
		if (cards[i] == num)
			found = true;
	}
```

We can place this code in a loop so that it repeats indefinitely (or until the user enters some "stopping" number such as -1):

```
import java.util.Scanner;

public class CardSearchProgram2 {
	public static void main(String[] args) {
		boolean found;
		int[]   cards = {2, 4, 9, 11, 12, 43, 45, 65, 76, 13, 84, 92, 95, 104};

		int num = 1;
		while (num > 0) {
			found = false;
			System.out.println("Which card are you looking for ?");
			num = new Scanner(System.in).nextInt();

			for (int i=0; i<cards.length; i++) {
				if (cards[i] == num)
					found = true;
			}
			if (found)
				System.out.println("Card #" + num + " is in the set.");
			else
				System.out.println("Card #" + num + " is not in the set.");
		}
	}
}
```

Notice that the **while** loop goes until 0 or negative numbers are entered. Also, the **num** variable is defined outside of the loop because we need to give it a value before getting into the loop. Lastly, note that the boolean flag **found** must be reset to **false** each time during the loop.

Example: (searching two lists at the same time and quitting when found)

Assume now that we have a deck of regular playing cards and we want to determine whether or not a specific card is in the deck and then quit as soon as we find it, rather than continuing to check all the remaining cards. A card is identified as having a **rank** (i.e., 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace) as well as a **suit** (i.e., Hearts, Diamonds, Spades, Clubs). We can represent the ranks as numbers from 2 through 13 and the suits as characters 'H', 'D', 'S', and 'C'.

The code is similar to before, but now we use two arrays ... one to store the ranks of the cards and the other to store the suits:

```
import java.util.Scanner;
public class CardSearchProgram3 {
	public static void main(String[] args) {
		boolean found = false;

		int[]  ranks = {2, 4, 5, 7, 9, 11, 12, 3, 6,
				10, 13, 7, 9, 3, 8, 12, 13};
		char[]  suits = {'H','H','H','H','H','H','H','D', 'D',
				'D','D','S','S','C','C','C','C'};
```

```
            System.out.println("What is the card's rank (2-14)?");
            int rank = new Scanner(System.in).nextInt();
            System.out.println("What is the card's suit (H, D, S, C)?");
            char suit = new Scanner(System.in).next().charAt(0);

            for (int i=0; i<ranks.length; i++) {
                  if ((ranks[i] == rank) && (suits[i] == suit))
                        found = true;
            }
            if (found)
                  System.out.println("Card #"+rank+" of "+suit+" is in there");
            else
                  System.out.println("Card #"+rank+" of "+suit+" is not in there");
      }
}
```

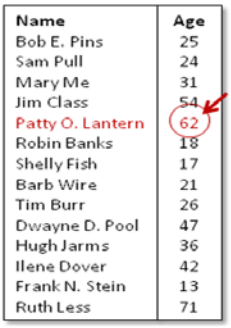
The code is very similar to before, but now accesses both arrays to make sure that the suit AND rank both match. The above code is not very efficient. A standard deck of cards contains 52 cards (without joker cards). What if the very first card was the one that we were looking for ? The code above will continue to check the remaining 51 cards. That is not very efficient. We can use the **break** statement to tell Java that we want to "break out" of the loop. Here is how it would be used in this scenario:

```
      for (int i=0; i<ranks.length; i++) {
            System.out.println("Checking card: " + i);      // just to verify
            if ((ranks[i] == rank) && (suits[i] == suit)) {
                  found = true;
                  break;
            }
      }
      if (found)
            System.out.println("Card #"+rank+" of "+suit+" is in there");
      else
            System.out.println("Card #"+rank+" of "+suit+" is not in there");
```

When the card is found, the above code will stop the **for** loop and continue with the next line after the loop ... which is the **`if (found)`** statement. You can verify this by observing the indices printed at the beginning of the loop. Now this is much more efficient (up to 51 times faster for a full deck of cards, but just 2 times faster on average).

Example: (searching for a String and returning a corresponding attribute)

Consider now trying to look through a list to determine someone's age. Assume that it is just a list of names and corresponding ages. We can store the names in an array of Strings and the ages in an equal-sized array of ints.



Name	Age
Bob E. Pins	25
Sam Pull	24
Mary Me	31
Jim Class	54
Patty O. Lantern	62
Robin Banks	18
Shelly Fish	17
Barb Wire	21
Tim Burr	26
Dwayne D. Pool	47
Hugh Jarms	36
Ilene Dover	42
Frank N. Stein	13
Ruth Less	71

To create an array of names, we set the type to be `String[]`, which is an array of String objects.

Here is how to do this:

```
String[]  names = {"Bob E. Pins", "Sam Pull",
                   "Mary Me", "Jim Class",
                   "Patty O. Lantern", "Robin Banks",
                   "Shelly Fish", "Barb Wire",
                   "Tim Burr", "Dwayne D. Pool",
                   "Hugh Jarms", "Ilene Dover",
                   "Frank N. Stein", "Ruth Less"};
```

We would want to have a corresponding array of ages:

```
int[]  ages = {25, 24, 31, 54, 62, 18, 17, 21, 26, 47, 36, 42, 13, 71};
```

This array should be the same size so that each name has a corresponding age at the same position in the array. We would then need to ask which name is being searched for and get it from the user as follows:

```
inputName = new Scanner(System.in).nextLine();
```

Finally, one more note about Strings, is that to compare them, we cannot use the **==** sign, but instead must use a function called **equals()**. We will discuss why this is so, later in the course. So here is how we would compare a name in the list to the one entered:

```
if (names[i].equals(inputName)) //...
```

All that remains to do then, is to put it all together. Remember ... we are looking for the age of someone, and we want to print it out. So, we will probably want to remember the age in a variable as we go through the list so that we can print it out later. Here is the completed code:

```
import java.util.Scanner;

public class AgeSearchProgram {
      public static void main(String[] args) {

            int      age = 0;
            String   inputName;

            String[]  names = {"Bob E. Pins", "Sam Pull", "Mary Me", "Jim Class",
                               "Patty O. Lantern", "Robin Banks", "Shelly Fish",
                               "Barb Wire", "Tim Burr", "Dwayne D. Pool",
                               "Hugh Jarms", "Ilene Dover", "Frank N. Stein",
                               "Ruth Less"};

            int[]  ages = {25, 24, 31, 54, 62, 18, 17, 21, 26, 47, 36, 42, 13, 71};

            System.out.println("Whose age are you looking for ?");
            inputName = new Scanner(System.in).nextLine();

            for (int i=0; i<names.length; i++) {
                  if (names[i].equals(inputName)) {
                        age = ages[i];
                        break;
                  }
            }
```

```
            if (age != 0)
                  System.out.println("The age of " + inputName + " is " + age);
            else
                  System.out.println(inputName + " is not on the list.");
      }
}
```

Example: (searching for a maximum/minimum)

Another very common task when searching through a set of values is to find the maximum or minimum. Can you write a program to find the oldest and the youngest people on a list ? Let us see if we can do this with functions as well. The functions should take, as parameters, an array of names and a corresponding array of ages. They should then return the name of the oldest/youngest person. Here is what the functions should look like:

```
      public static String findOldestPerson(String[] names, int[] ages) {
            // ...
      }
      public static String findYoungestPerson(String[] names, int[] ages) {
            // ...
      }
```

Notice that the return type is a String (because it is a person's name) and that the array can be passed in as parameters just like any other values, provided that the square brackets [] are used. Here is the structure for the program:

```
import java.util.Scanner;

public class OldestYoungestProgram {

      public static String findOldestPerson(String[] names, int[] ages) {
            // ... code not completed yet ...
      }
      public static String findYoungestPerson(String[] names, int[] ages) {
            // ... code not completed yet ...
      }

      public static void main(String[] args) {
            String[]  names = {"Bob E. Pins", "Sam Pull", "Mary Me", "Jim Class",
                               "Patty O. Lantern", "Robin Banks", "Shelly Fish",
                               "Barb Wire", "Tim Burr", "Dwayne D. Pool",
                               "Hugh Jarms", "Ilene Dover", "Frank N. Stein",
                               "Ruth Less"};
            int[]  ages = {25, 24, 31, 54, 62, 18, 17, 21, 26, 47, 36, 42, 13, 71};

            System.out.println("The oldest person is " +
                                          findOldestPerson(names, ages));
            System.out.println("The youngest person is " +
                                          findYoungestPerson(names, ages));
      }
}
```

How do we write the function now ? Well, we simply treat the incoming parameters as regular variables. We can use a **for** loop to determine the maximum age and then return the name of the person with that age. In order to determine the oldest person's age, this is like finding a maximum value from a list. We will need to keep track of the "oldest person so far" as we are going through the list ... which is the person with the "maximum age so far". Whenever we find a person whose age is larger than the "maximum age so far" then that person's age becomes the "maximum age so far". Here is what we have so far:

```
public static String findOldestPerson(String[] names, int[] ages) {
      int   maximumAgeSoFar = 0;
      for (int i=0; i<ages.length; i++) {
            if (ages[i] > maximumAgeSoFar)
                  maximumAgeSoFar = ages[i];
      }
      // ... more to go ...
}
```

When the **for** loop ends, we will have the age of the oldest person! But, we need to know the name of the oldest person. Unfortunately, we are just holding on to the actual age ... and by the time the loop ends, we don't know who that person was.

There are two simple ways to fix this. The first one is to keep track of the oldest person's **name** as well as the **age**. We would just need another variable. Then we know the person to return from the function:

```
public static String findOldestPerson(String[] names, int[] ages) {
      int         maximumAgeSoFar = 0;
      String      oldestPersonSoFar;
      for (int i=0; i<ages.length; i++) {
            if (ages[i] > maximumAgeSoFar) {
                  maximumAgeSoFar = ages[i];
                  oldestPersonSoFar = names[i];
            }
      }
      return oldestPersonSoFar;
}
```

A different (and better) way to do this is to just keep track of the **index** position of the oldest person so far instead of the person's age and name. The following code starts off assuming that the first person in the list is the oldest person (i.e., at position 0).

```
public static String findOldestPerson(String[] names, int[] ages) {
      int   oldestPersonSoFar = 0;
      for (int i=0; i<ages.length; i++) {
            if (ages[i] > ages[oldestPersonSoFar])
                  oldestPersonSoFar = i;
      }
      return names[oldestPersonSoFar];
}
```

What about finding the youngest person ? The code is similar ... but we just need to change the comparison:

```
public static String findYoungestPerson(String[] names, int[] ages) {
      int   youngestPersonSoFar = 0;
      for (int i=0; i<ages.length; i++) {
            if (ages[i] < ages[youngestPersonSoFar])
                  youngestPersonSoFar = i;
      }
      return names[youngestPersonSoFar];
}
```

However, this will not work if the array has size zero (which is a weird situation that should never happen, but someone may possibly make an error somewhere and a zero-sized array may get passed in as a parameter). Just be careful ! What should the answer be if there are no people in the array ? We can always return an empty string as a value ... or null. Null represents an "undefined object" ... and this is often used as a kind of "*invalid solution*" return value for functions that require an object to be returned:

```
public static String findYoungestPerson(String[] names, int[] ages) {
      if (ages.length == 0)
            return null;  // or return "";

      int   youngestPersonSoFar = 0;
      for (int i=1; i<ages.length; i++) {
            if (ages[i] < ages[youngestPersonSoFar])
                  youngestPersonSoFar = i;
      }
      return names[youngestPersonSoFar];
}
```

5.3 Comparing and Selecting Items in Arrays

This section contains some more example programs which use arrays to handle situations such as checking the items in one or more arrays to see whether they meet a specific criteria, comparing two arrays with one another, extracting items from an array, and splitting arrays.

Example: (checking a list to see if it is in order)

Suppose that we had a stack of a few trading cards and that we wanted to know if the cards were in order sequentially, based on the unique number that each card has on the back. Write a program that goes through an array of these card numbers and determines whether or not the cards are in increasing order.

To begin, here is what we will start with:

```
public class CardsInOrderProgram {
	public static void main(String[] args) {
		boolean inOrder = true;

		int[]  cards = {2, 4, 9, 11, 12, 43, 45, 65, 76, 13, 84, 92, 95, 104};

		// ... complete the code ...

		if (inOrder)
			System.out.println("The cards are in order.");
		else
			System.out.println("The cards are not in order.");
	}
}
```

Again, we are using a boolen flag. We will assume that the cards are in order and simply set the boolean variable to **false** if the cards are found to be out of order. Now, how do we solve the problem of determining whether the cards are in order ? Well, we need to go through each number in order and make sure that the numbers before it are all smaller.

Here is one strategy: Keep track of the maximum number and make sure that each number is larger than the maximum. If ever one is smaller than the maximum, it was out of order.

```
	int   maximum = 0;
	for (int i=0; i<cards.length; i++) {
		if (cards[i] < maximum)
			inOrder = false;
		else
			maximum = cards[i];
	}
```

A different way of doing this is as follows. Instead of keeping a **maximum** variable, we can just check adjacent pairs of numbers to make sure that the one on the right is larger than the one on the left.

```
public class CardsInOrder2Program {
	public static void main(String[] args) {
		boolean inOrder = true;

		int[]  cards = {2, 4, 9, 11, 12, 43, 45, 65, 76, 13, 84, 92, 95, 104};

		for (int i=1; i<cards.length; i++) {
			if (cards[i-1] > cards[i])
				inOrder = false;
		}

		if (inOrder)
			System.out.println("The cards are in order.");
		else
			System.out.println("The cards are not in order.");
	}
}
```

How can we make a function that takes the array as a parameter and determines whether or not the cards are in order ? Well, the array can be passed in as any other variable and a boolean value must be returned. Note the use of the **return** statement:

```
public class CardsInOrder3Program {

	public static boolean inOrder(int[] array) {
		for (int i=1; i<array.length; i++) {
			if (array[i-1] > array[i])
				return false;
		}
		return true;
	}

	public static void main(String[] args) {
		int[]  cards1 = {2, 4, 9, 11, 12, 43, 45, 65, 76, 13, 84, 92, 95, 104};
		int[]  cards2 = {2, 4, 9, 11, 12, 13, 43, 45, 65, 76, 84, 92, 95, 104};
		int[]  cards3 = {};

		if (inOrder(cards1))
			System.out.println("Card set 1 is in order.");
		else
			System.out.println("Card set 1 is not in order.");

		if (inOrder(cards2))
			System.out.println("Card set 2 is in order.");
		else
			System.out.println("Card set 2 is not in order.");

		if (inOrder(cards3))
			System.out.println("Card set 3 is in order.");
		else
			System.out.println("Card set 3 is not in order.");
	}
}
```

Note that since we have a nice function now, we can call it easily with different card sets. Interestingly, our solution even works for an empty array.

Example: (comparing contents of two or more arrays)

Assume now that we have three hands of 5 playing cards each and we want to know which is the "higher" hand in terms of points. We would just need to look at the ranks of the cards and compare their totals. Write a function that compares two hands (as incoming **int** arrays of size 5) and returns true if the first hand is higher in total, otherwise the 2nd is returned.

```
public class HighestHandProgram {

	// Returns true if the 1st hand is higher in total than the 2nd hand
	public static boolean isHigher(int[] hand1, int[] hand2) {
		int   total1 = 0, total2 = 0;

		for (int i=0; i<hand1.length; i++) {
			total1 += hand1[i];
			total2 += hand2[i];
		}
		return total1 > total2;
	}

	public static void main(String[] args) {
		int[]  cards1 = {2, 7, 4, 9, 13};
		int[]  cards2 = {6, 1, 8, 7, 10};
		int[]  cards3 = {3, 2, 12, 9, 11};

		if (isHigher(cards1, cards2))
			if (isHigher(cards1, cards3))
				System.out.println("Hand 1 is the highest");
			else
				System.out.println("Hand 3 is the highest");
		else
			if (isHigher(cards2, cards3))
				System.out.println("Hand 2 is the highest");
			else
				System.out.println("Hand 3 is the highest");
	}
}
```

Example: (selecting multiple values from an array that match a criteria)

Another very common task when searching through a set of values is to select a bunch of values from the array. For example, given an array of numbers, let us write a function that returns a new array which contains all the numbers from the original array that are odd numbers. This can be useful for selecting roughly half of a set of numbers from a random set. For example, if we had 20,000 students, roughly half of their student numbers will be even, the other half will be odd. So, selecting the odd ones would roughly select half of the students from the list in a fair manner (as opposed, for example, by splitting into males/females or alphabetically A-L and M-Z as is often done in life). Here is what the function should look like:

```
public static int[] selectOdd(int[] studentNumbers) {
	// ...
}
```

Notice that the return type is an array of integers. This is not the same array that we started with. It is a **new** array which will contain the odd numbers from the **studentNumbers** array. How big should this array be ? In theory, all student numbers could be odd ... and so it may need to be as big as the original:

```
int[] oddNumbers = new int[studentNumbers.length];
```

Then we just need to loop through and fill it up with the odd numbers. Odd numbers will not be divisible by two. So, we can use the modulus operator (%) to find them. If we modulo the numbers by 2 (i.e., find the remainder after dividing by two), the result will always be either 0 or 1. If the result is 1, the number was odd, otherwise it was even. Here is the structure:

```
for (int i=0; i<studentNumbers.length; i++) {
      if (studentNumbers[i]%2 == 1)
            // Add this number to the result
}
```

Now, how do we "add the number to the result" ? Well, we are "filling in" the **oddNumbers** array as we find odd numbers. Recall that to put something into an array, we need to specify the index position. So, we will need to keep track of where we are in that array that we are filling up so that we can put the items in consecutive positions in the array. All we need is a counter to keep track of how many items we have in there so far. Here is the result:

```
public static int[] selectOdd(int[] studentNumbers) {
      int[] oddNumbers = new int[studentNumbers.length];
      int   oddNumbersFound = 0;

      for (int i=0; i<studentNumbers.length; i++) {
            if (studentNumbers[i]%2 == 1)
                  oddNumbers[oddNumbersFound++] = studentNumbers[i];
      }
      return oddNumbers;
}
```

Notice as well that we MUST NOT FORGET to increase the **oddNumbersFound** counter after we add the odd number to the result. That way, the next item will go into the next position and will not overwrite the one we just put in !

```
public class OddStudentNumberProgram {

      public static int[] selectOdd(int[] studentNumbers) {
            int[] oddNumbers = new int[studentNumbers.length];
            int   oddNumbersFound = 0;

            for (int i=0; i<studentNumbers.length; i++) {
                  if (studentNumbers[i]%2 == 1)
                        oddNumbers[oddNumbersFound++] = studentNumbers[i];
            }
            return oddNumbers;
      }

      public static void main(String[] args) {
            int[]  nums = {162793, 170983, 177914, 276385, 167822, 181830,
                           278924, 178962, 187923, 127891, 128936, 179128};

            System.out.println("Here are the odd numbers:");

            int[] result = selectOdd(nums);
            for (int i=0; i<result.length; i++)
                  System.out.println(result[i]);
      }
}
```

This code will work. However, notice the result:

```
Here are the odd numbers:
162793
170983
276385
187923
127891
0
0
0
0
0
0
0
```

The array has the same size as the original array, which is 12. But there were only 5 odd student numbers in the array, so only the first 5 positions will filled in. The remaining positions still have zero in them. In this scenario, it is not that big of a concern, because the zeros are clearly not student numbers. So we could fix this in the code as follows:

```
	for (int i=0; i<result.length; i++) {
		if (result[i] != 0)
			System.out.println(result[i]);
	}
```

Since each value is checked before displaying, the zeros will not be displayed. However, after encountering the first zero, the loop still continues checking all the remaining numbers ... which are all zeros. A more efficient way to do this check is to break out of the loop once the first zero has been identified:

```
	for (int i=0; i<result.length; i++) {
		if (result[i] == 0)
			break;
		System.out.println(result[i]);
	}
```

This is now more time-efficient. However, the code is still not space-efficient. If there were 20,000 student numbers and half were odd, we will have an array of size 20,000 but with 10,000 valid numbers and 10,000 zeros. Each **int** takes 4 bytes of memory, so we are allocating (i.e., wasting) **40 kilobytes** of memory without necessity. That is poor programming style. How can we fix this ?

Well, we need to allocate just enough space for the answer. So, before creating the array, we need to determine how many odd numbers there are. Then we can make the array the exact size that is needed.

This will require an extra **for** loop:

```
public static int[] selectOdd(int[] studentNumbers) {

      // First determine the number of odd numbers in the array
      int   oddNumbersFound = 0;
      for (int i=0; i<studentNumbers.length; i++) {
            if (studentNumbers[i]%2 == 1)
                  oddNumbersFound++;
      }

      // Now create and fill-up the array
      int[] oddNumbers = new int[oddNumbersFound];

      oddNumbersFound = 0;    // reset
      for (int i=0; i<studentNumbers.length; i++) {
            if (studentNumbers[i]%2 == 1)
                  oddNumbers[oddNumbersFound++] = studentNumbers[i];
      }

      return oddNumbers;
}
```

And voila! Our code is now space-efficient. However, it came at a cost of being a little slower, because we have to search through the numbers twice. In your life as a computer scientist, you will learn that there is often a trade-off between time efficiency and space efficiency. The trick is to get the right balance for the problem at hand. This tradeoff is common in real life. For example, if you wanted to sort 250 exam papers by grade, it would take a long time to do this sitting on a chair with them on your lap. However, if you had a lot more space available (i.e., a large empty table in front of you), then you could make use of that to make the sorting go much quicker (e.g., sort into smaller piles).

Example: (dividing array data into multiple individual arrays)

The above example extracted the odd numbers from an array. How could we extract two arrays ... one for the odd numbers and another for the even numbers ? Since functions require one return type, we cannot return two things. However, there are a few ways to solve the problem ... here are two:

(1) We could write two methods. One to extract the odd, the other to extract the even.
(2) We could create the two arrays in advance and pass them in to the function.

The 2nd strategy is interesting. We can adjust the function to take three arrays as parameters (the original array, the array that will contain the odd numbers and the array that will contain the even numbers). There need not be a return type anymore, since there is no particular result being returned.

Instead, the arrays are simply filled in with the appropriate data:

```
	public static void extractOddEven(int[] studentNumbers, int[] oddNumbers,
                                                                int[] evenNumbers) {
		int   oddNumbersFound = 0, evenNumbersFound = 0;
		for (int i=0; i<studentNumbers.length; i++) {
			if (studentNumbers[i]%2 == 1)
				oddNumbers[oddNumbersFound++] = studentNumbers[i];
			else
				evenNumbers[evenNumbersFound++] = studentNumbers[i];
		}
	}
```

Of course, to test it, we need to create the arrays in advance:

```
public class OddEvenStudentNumberProgram {
	public static void extractOddEven(int[] studentNumbers, int[] oddNumbers,
                                                                int[] evenNumbers) {
		// ... code written as above ... omitted here to save space
	}

	public static void main(String[] args) {
		int[]  nums = {162793, 170983, 177914, 276385, 167822, 181830,
                               278924, 178962, 187923, 127891, 128936, 179128};

		// Create the arrays that will be "filled-up"
		int[] oddArray = new int[nums.length];
		int[] evenArray = new int[nums.length];

		// Extract the data into the two arrays
		extractOddEven(nums, oddArray, evenArray);

		// Display the results
		System.out.println("Here are the odd numbers:");
		for (int i=0; i<oddArray.length; i++) {
			if (oddArray[i] == 0)
				break;
			System.out.println(oddArray[i]);
		}
		System.out.println("Here are the even numbers:");
		for (int i=0; i<evenArray.length; i++) {
			if (evenArray[i] == 0)
				break;
			System.out.println(evenArray[i]);
		}
	}
}
```

This is an example of parameters that are *pass-by-reference*. This means, that we are passing a "pointer" to the array's memory location (i.e., reference). This way, within the procedure, we have the array itself ... so we can access and modify it during the procedure. When the procedure has completed, the arrays have changed.

5.4 Dealing With Array Capacity and Unwanted Values

Since arrays are fixed in size, we cannot make them bigger once they get full. Sometimes, when we are not sure of how big we need an array, we may chose a somewhat arbitrary value as the array's maximum capacity. Ultimately, however, it is possible that we may attempt to add an item beyond the array's capacity. In such a case, the program will usually crash (or stop unexpectedly and non-gracefully). Often, we get an unpleasant Exception:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 10000
```

In addition to exceeding an array's capacity, sometimes we need to remove data from an array because it is no longer valid. However, since we cannot shrink an array, we can only remove data by placing a piece of garbage data in its place ... a kind of ***overwrite*** at that location in the array. As time goes on, the array can be filled with much garbage data and it could slow down searching algorithms as well as the processes of adding new pieces of data.

How do we address these potentially serious problems ?

For the first issue, since arrays are fixed size, we cannot simply make more room within the existing array. Rather, a new *bigger* array must be created and all elements must be copied into the new array. But how much bigger ? It's up to us.

Regarding the second issue, we need to ***defragment*** (or ***consolidate***) the data by bringing all the data together again into contiguous (one after another) memory locations. Let us look at some examples.

Example:

Consider a program that continually asks for integers from the user, and adds them to an array as they come in until **-1** is entered, and then does something interesting with the numbers that were entered:

```
import java.util.Scanner;
public class ReadInNumbersProgram {
	public static void main(String[] args) {
		int[]		numbers = new int[100];
		int		count = 0;
		int		enteredValue = 0;

		while (enteredValue != -1) {
			System.out.print("Enter number " + (count+1) + ": ");
			enteredValue = new Scanner(System.in).nextInt();
			if (enteredValue != -1)
				numbers[count++] = enteredValue;
		}
		// ... do something interesting with the numbers ...
	}
}
```

Notice that each number coming in is added to the array according to the **count** position, which is incremented each time a valid number arrives. When the loop has completed, **count** represents the number of integers that were entered and added to the array. The code is straight forward. However, can you foresee a problem that may arise ?

What happens when we try to enter too many numbers ? After all, the user has no idea that the maximum amount of numbers that can be entered is 100. This code will crash when the 101st number is entered.

Clearly, it is simple to increase the capacity of the array to 200, 500, 10000 or whatever. But what if only 105 numbers are needed ? Do we really want to create a huge array, allocating space that we won't need ? No. Even if we do pick a large value, it is still possible that we may exceed it (e.g., in scenarios where we are reading data from a file...because nobody in their right mind would sit and type in 10000 integers into a program).

Instead, an option that we could take is to simply increase the storage by some incremental value each time we are about to exceed the limit. We can increase by some constant value such as 5, 10, 50, 100, etc... To do this, we would need to make an entirely new array which has a bigger size, and then copy into it all of the values that we have read in so far (there are more efficient ways to deal with this, but let us take this approach for now).

Here is a function that will create a bigger array by some given amount and return it:

```
public static int[] enlarge(int[]  originalArray, int amountToIncreaseBy) {
    // First, create the bigger array
    int[]   enlargedArray = new int[originalArray.length + amountToIncreaseBy];

    // Now copy over all the values
    for (int i=0; i<originalArray.length; i++)
        enlargedArray[i] = originalArray[i];

    return enlargedArray;
}
```

Do you understand the code ? We can call this method any time that we need to "enlarge" the array. Keep in mind that we are not really *enlarging* the array, but we are making a new array that is bigger. So, we will need to remember to store the returned value from the function. Here is our "fixed" code now:

```
import java.util.Scanner;

public class ReadInNumbersProgram2 {
    public static final int         GROWTH_SIZE = 5;  // Amount to grow array by

    // Function to make the array bigger
    public static int[] enlarge(int[]  originalArray, int amountToIncreaseBy) {
        int[]   enlargedArray = new int[originalArray.length + amountToIncreaseBy];

        for (int i=0; i<originalArray.length; i++)
            enlargedArray[i] = originalArray[i];

        return enlargedArray;
    }
```

```
    public static void main(String[] args) {
        int[]   numbers = new int[5];
        int     count = 0;
        int     enteredValue = 0;

        while (enteredValue != -1) {
            System.out.print("Enter number " + (count+1) + ": ");
            enteredValue = new Scanner(System.in).nextInt();
            if (enteredValue != -1) {
                if (count == numbers.length) {
                    // Change the numbers variable to point to the new/bigger array
                    numbers = enlarge(numbers, GROWTH_SIZE);
                }
                numbers[count++] = enteredValue;
            }
        }

        // Print out the array, as a test
        for (int i=0; i<numbers.length; i++)
            System.out.print(numbers[i] + ", ");
        System.out.println();
    }
}
```

We can run the program a few times to see if it works:

```
Enter number 1: 1
Enter number 2: 2
Enter number 3: -1
1, 2, 0, 0, 0,                                  // Notice that it is still size 5
```

```
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
Enter number 4: 4
Enter number 5: 5
Enter number 6: 6
Enter number 7: -1
1  2  3  4  5  6  0  0  0  0                    // It has grown to size 10 now
```

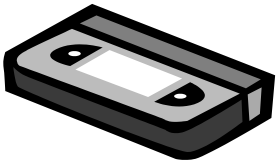
```
Enter number 1: 1
Enter number 2: 2
Enter number 3: 3
Enter number 4: 4
Enter number 5: 5
Enter number 6: 6
Enter number 7: 7
Enter number 8: 8
Enter number 9: 9
Enter number 10: 10
Enter number 11: 11
Enter number 12: 12
Enter number 13: -1
1  2  3  4  5  6  7  8  9  10  11  12  0  0  0              // It has grown to size 15 now
```

Example:

Consider having two same-size arrays ... one of names and another of the ages of the people with those names. Here are the arrays that we used previously:

```
String[]  names = {"Bob E. Pins", "Sam Pull", "Mary Me",
                   "Jim Class", "Patty O. Lantern",
                   "Robin Banks", "Shelly Fish",
                   "Barb Wire", "Tim Burr", "Dwayne D. Pool", "Hugh Jarms",
                   "Ilene Dover", "Frank N. Stein", "Ruth Less"};
int[]  ages = {25, 24, 31, 54, 62, 18, 17, 21, 26, 47, 36, 42, 13, 71};
```

Now, assume that we need to discard (or remove from the arrays) all people below the age of 21. In the above example, there are three such people.



How could we do this ? Well, each array has a fixed size, so we cannot really "remove" any data ... the array will never get smaller. The situation is analogous to a program recorded on a videotape, we cannot actually remove the item but we can only overwrite it (i.e., replace it) with a new value.

So, to delete a piece of information from an array, we would need to replace it with some value that is considered invalid data. Perhaps an age of **-1** or a name which is either an empty string **""** or **null**. The array will stay the same size, but the original data will be gone. Here is code that will do what we want:

```
for (int i=0; i<ages.length; i++) {
      if (ages[i] < 21) {
            ages[i] = -1;
            names[i] = null;
      }
}
```

This will ensure that all names and ages of people under the age of 21 are deleted. Afterwards, however, there will be some locations in the array where the ages are -1 and names are **null** as can be seen here. These are like "holes" in the array.

After

	names	ages
0	"Bob E. Pins"	25
1	"Sam Pull"	24
2	"Mary Me"	31
3	"Jim Class"	54
4	"Patty O. Lantern"	62
5	null	-1
6	null	-1
7	"Barb Wire"	21
8	"Tim Burr"	26
9	"Dwayne D. Pool"	47
10	"Hugh Jarms"	36
11	"Ilene Dover"	42
12	null	-1
13	"Ruth Less"	71

These "holes" may present two problems. First, we don't know how many people are left in the array. Second, when looping through the array we will encounter **null** objects that we need to deal with properly.

To handle the issue regarding the amount of valid data remaining in the array, we can always maintain a variable indicating the number of such valid elements as follows:

```
int validPeople = 0;

for (int i=0; i<ages.length; i++) {
      if (ages[i] < 21) {
            ages[i] = -1;
            names[i] = null;
      }
      else
            validPeople++;
}
```

This **validPeople** variable will indicate the number of people who are 21 or over.

Now, although we know how many valid entries are in the array, we still do not know the positions of these valid entries. Therefore, each time that we go through the array, we need to consider the possibility that there can be a "hole" at any position in the array. So we need to check for this each time. For example, if we want to print out the people and their ages, we would like to do this:

```
for (int i=0; i<ages.length; i++) {
      System.out.println(ages[i] + " year old " + names[i]);
}
```

but unfortunately, we now have to check for "holes":

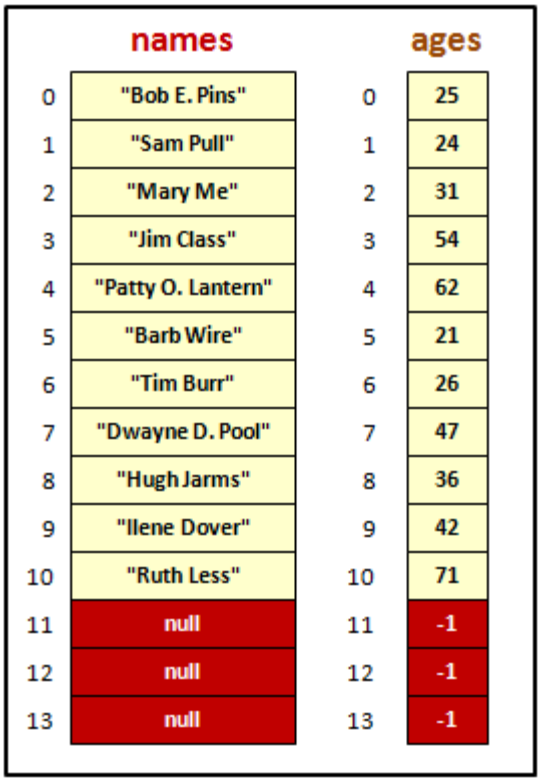
```
for (int i=0; i<ages.length; i++) {
      if (ages[i] > 0) { // or if (names[i] != null)
            System.out.println(ages[i] + " year old " + names[i]);
      }
}
```

However, it is often unpleasant to continually check for “holes” in the array like this. In fact, in a typical application we are likely more often going to need to traverse through elements of an array for display purposes than to search through the array to remove data. Therefore, it would be more advantageous to “fill-in” the hole each time so that the valid array elements are at the front-most part of the array at all times.

Assume that we have such a valid array in which all the elements are at the front-most part of the array. Assume also that we always have a count as to how many people are stored in the array at all times.

We will need to always have a variable associated with the array to indicate its size. It works the same as the **validPeople** variable, but we will call it something more general such as **numPeople**. In the picture here, **numPeople** would have a value of **11**.

Cleaner Version



We will ensure that all valid items in the array would be in positions **0** to **numPeople -1**, inclusively ... and all positions from **numPeople** to the array's **length -1** will contain garbage data (e.g., **null** or -1).

Consider a procedure that takes the name and age of a person and adds them to the existing arrays:

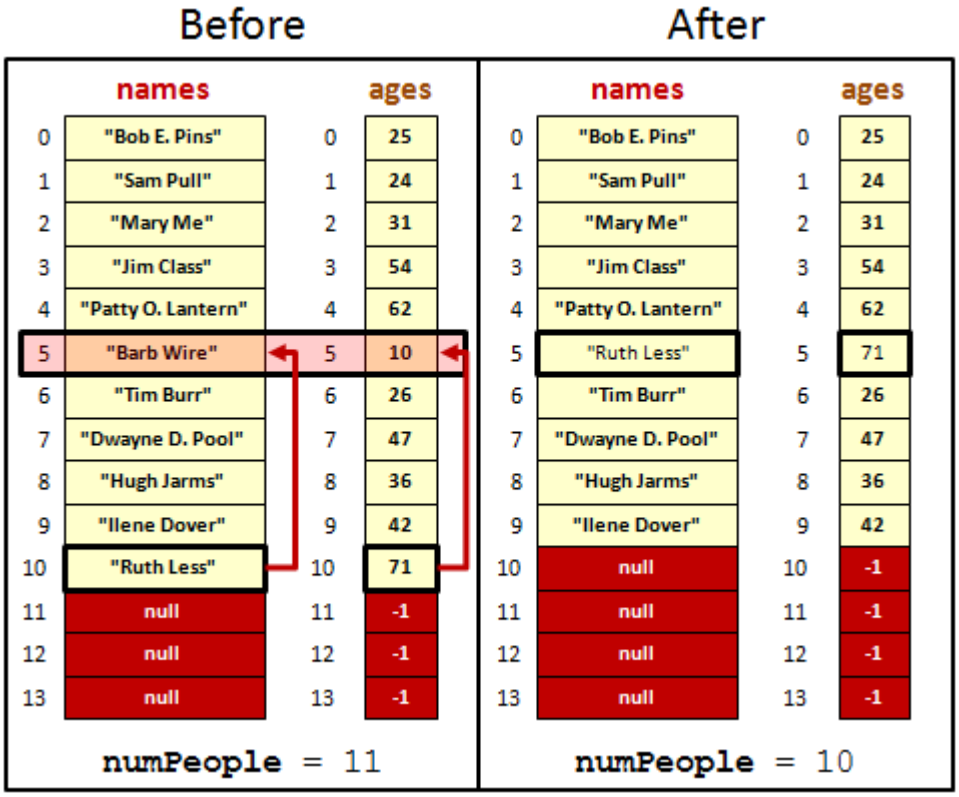
```
public static void add(String newName, int newAge) {
      // Put the person in the arrays
      if (numPeople < names.length) {
            names[numPeople] = newName;
            ages[numPeople] = newAge;
            numPeople++;
      }
}
```

First we make sure that there is room, and then simple add the person to the end of the arrays, making sure to increase the total count of **numPeople**. This code assumes that the arrays and the **numPeople** counter are defined as static variables. How though do we remove a person ? Well, could simply move the last person in the array to fill-in the "hole" that is created by the deletion of the person. Then erase the data from the last position in the array:

```
public static void removeUnderagePeople() {
      for (int i=0; i<numPeople; i++) {
            if (ages[i] < 21) {
                  ages[i] = ages[numPeople-1];
                  names[i] = names[numPeople-1];
                  ages[numPeople-1] = -1;
                  names[numPeople-1] = null;
                  numPeople--;
            }
      }
}
```

We need to make sure that we reduce our counter by one to indicate that we have one less piece of data.

However, it is possible that the last person in the array that we try to move over is also underage. Therefore, we could go ahead and move it over, but make sure to check the age of the person that we moved over as well before we move on to the next index in the

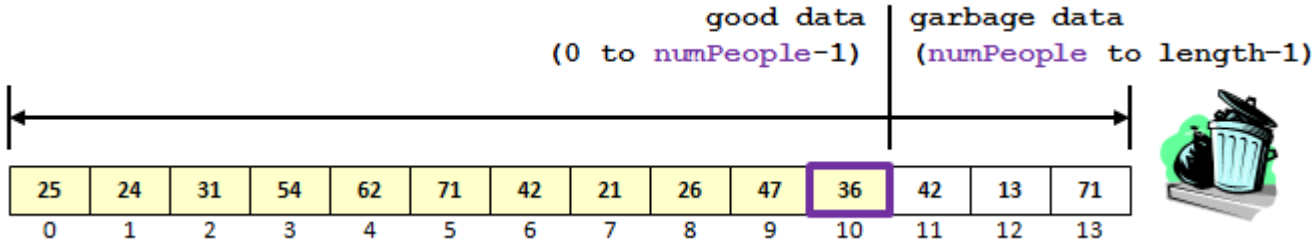


array. We just need to make sure that we do not move to the next index in the loop until we also check the location that we just inserted into. Basically, we can just subtract one from the loop index so that when we go to the next round of the loop, the **i++** will counter act the **i--**, leaving **i** as the same value that we just checked. That will ensure that the "hole" position is checked a second time, in case we moved an invalid person in there:

```
public static void removeUnderagePeople() {
      for (int i=0; i<numPeople; i++) {
            if (ages[i] < 21) {
                  ages[i] = ages[numPeople-1];
                  names[i] = names[numPeople-1];
                  ages[numPeople-1] = -1;
                  names[numPeople-1] = null;
                  numPeople--;
                  i--; // ensure that same position is checked another time
            }
      }
}
```

On the following page, there is a diagram showing what happens as the index **i** moves through the array. It shows how the items are inserted to fill the holes and how the **numPeople** variable decreases as the items are removed.

Note that it is not actually necessary to ***move*** the item to the open "holes". The item may simply be ***copied*** over, leaving "garbage" at the end of the array:

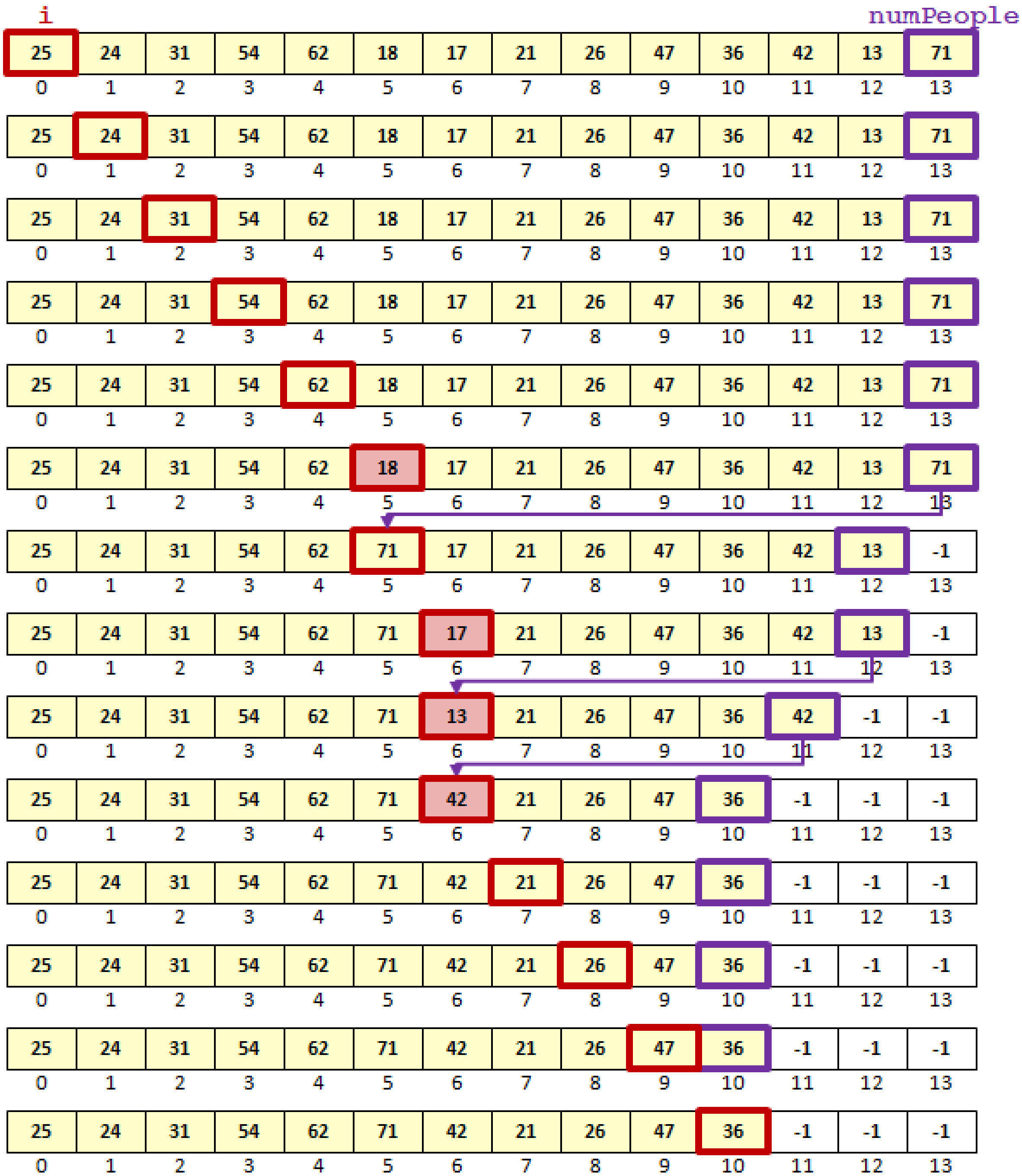


Homework:

The solution that we just came up with for keeping the data contiguous has one small issue ... it alters the ordering of the original data. That is, suppose that the data came in on a *first-come-first-served* basis. We want to ensure that we maintain the *fair* ordering of the data. With our current solution, after we remove the underage people, the original data order is lost. Try to re-do the **removeUnderagePeople()** method so that it maintains the items in the original order at all times. (see below)

original	25	24	31	54	62	18	17	21	26	47	36	42	13	71
solution	25	24	31	54	62	71	42	21	26	47	36	-1	-1	-1
homework	25	24	31	54	62	21	26	47	36	42	71	-1	-1	-1

Here is the diagram for the example that we just did (not the homework part):

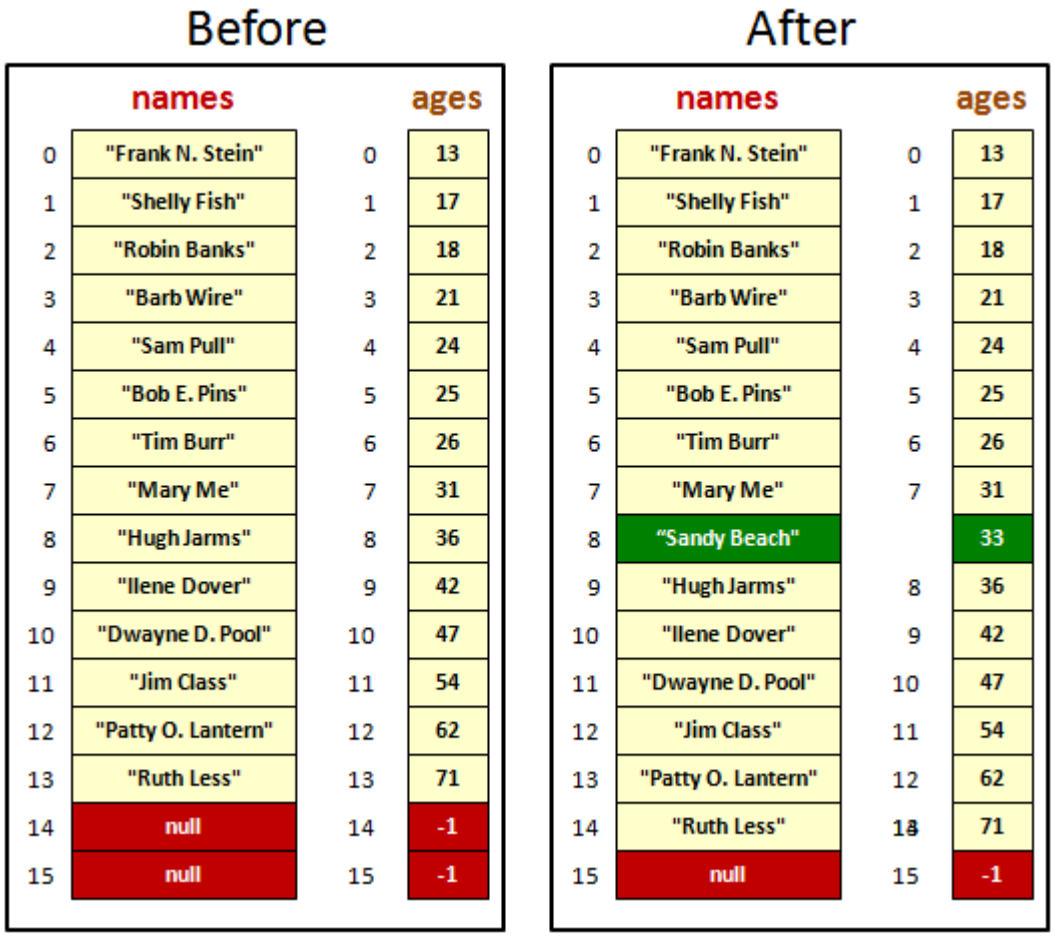


Example:

Assume that we already had a list of people sorted by their ages. How can we write code that will ***insert*** a new person into the appropriate location in the array so that it remains sorted ?

Well, to insert at some index position (e.g., **i**) we must shift all array items from indices **i** onwards one position further in the array:

For example, assume that we need to add "Sandy Beach" to the list who happens to be 33 years old. Here is what we need to do:



To complete this task, let us write a method called **insertPerson()** that takes the name and age of the person and adds them to the arrays, while keeping them sorted. First, we must determine the location to insert. To do this, we need to find the first person whose age is above the age of the new person. In the diagram above, this would be "Hugh Jarms" who is 36 years old at position 8 in the arrays. That will be the location to insert the new person. We can do this with a FOR loop:

```
public static void insertPerson(String newName, int newAge) {
      // Determine the position to insert the new person at in the arrays
      int   insertPosition = 0;
      for (int i=0; i<numPeople; i++) {
            if (ages[i] > newAge) {
                  insertPosition = i;
                  break;
            }
      }
}
```

Now we need to actually insert the person. However, we cannot simply place the **newName** and **newAge** at the specified position in the array because there is valid data there that would

be overwritten (i.e., lost) if we do that. First, we must move all the others down in the array, then we can insert. Do you understand what is wrong with this code ?

```
// Make space by moving the remaining data down
for (int i=insertPosition; i<numPeople-1; i++) {
      ages[i+1] = ages[i];
      names[i+1] = names[i];
}
```

This code will not produce what we want. The problem is that when we move "Hugh Jarms" down from position 8 to position 9, we erase "Ilene Dover". Then when we copy from position 9 to 10, we are copying "Hugh Jarms" again! The code will simply copy "Hugh Jarms" to all the array locations past index 8.

To fix this, we simply need to start shifting things down by starting from the bottom:

```
// Make space by moving the remaining data down
for (int i=numPeople; i>insertPosition; i--) {
      ages[i] = ages[i-1];
      names[i] = names[i-1];
}
```

Finally, we need to do the insertion, remembering to increase the array count. Here is the final version of the code:

```
public static void insertPerson(String newName, int newAge) {
      // Determine the position to insert the new person at in the arrays
      int   insertPosition = 0;
      for (int i=0; i<numPeople; i++) {
            if (ages[i] > newAge) {
                  insertPosition = i;
                  break;
            }
      }
      // Make space by moving the remaining data down
      for (int i=numPeople; i>insertPosition; i--) {
            ages[i] = ages[i-1];
            names[i] = names[i-1];
      }
      // Now add the person
      ages[insertPosition] = newAge;
      names[insertPosition] = newName;
      numPeople++;
}
```

Example:

Here is a tougher one now. Consider an autoshow, where there are many cars, each with some uniform color (we will assume no multi-colored cars). What if we were on our way to the autoshow but before we left, our roommate asked us (for some strange/unknown reason) to determine the most popular color of car at the autoshow. How do we approach the problem ?

Well think of real life. Assuming that there were hundreds of cars and that your memory is not perfect, you would likely bring with you a piece of paper (perhaps on a clipboard) so that you can keep track of the colors of cars that you find.

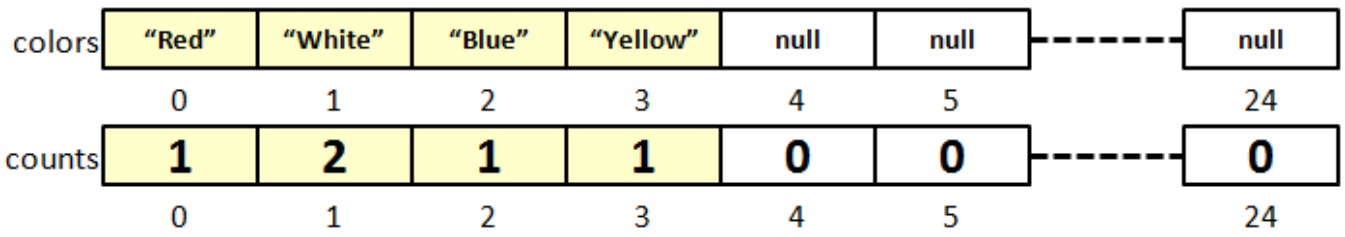
When you enter the autoshow and see the first car, you would look at its **color** and then likely write down the color on the paper and maybe put the number 1 beside it.

Assume that you went to the next car and that it was a different color. You would write that new color down too along with a count of 1. If you find a car with the same color again, you would just increment its count.

Below is a picture showing how your clipboard gets updated as you encounter car colors in the order of red, white, white, blue, etc..:

Let us assume that we have a program that lets us repeatedly enter car colors and it keeps track of theses colors in the manner above. As we enter the colors, each color will have a single number associated with it at all times (representing the **count** of the number of times that color appeared).

Since we need a list of *counts* along with the list of *colors*, we will need ***two*** arrays… one to store the **colors** and one to store the **count** for that color. But how big should we make the arrays ? Well, it depends on how many unique colors that we would expect to find. Let us set it at 25.



Here are the arrays and the counters for each that keep track of how many are in each array:

```
public static final int MAX_COLORS = 25;
String[]   colors = new String[MAX_COLORS];
int[]      counts = new int[MAX_COLORS];
int        uniqueColors = 0;
```

Here **uniqueColors** will increase each time we find a new color. Here is the basic code for getting the information from the user and storing it in the arrays as described:

```
import java.util.Scanner;

public class CarColorCountProgram {
	public static final int MAX_COLORS = 25;

	public static void main(String[] args) {
		String[]	colors = new String[MAX_COLORS];
		int[]		counts = new int[MAX_COLORS];
		int		uniqueColors = 0;

		System.out.print("Enter car color: ");
		String enteredColor = new Scanner(System.in).nextLine();

		// Keep going until no more colors are entered
		while (!enteredColor.equals("")) {
			// Determine if the color is already there
			int colorIndex = -1;
			for (int i=0; i<uniqueColors; i++) {
				if (colors[i].equals(enteredColor)) {
					colorIndex = i;
					break;
				}
			}
			// If the color is not there, then add it
			if (colorIndex == -1) {
				colorIndex = uniqueColors;
				colors[colorIndex] = enteredColor;
				uniqueColors++;
			}
			// Now add one to the color
			counts[colorIndex]++;

			// Get the next color from the user
			System.out.print("Enter car color: ");
			enteredColor = new Scanner(System.in).nextLine();
		}

		// Print out the arrays, as a test
		for (int i=0; i<uniqueColors; i++)
			System.out.println(colors[i] + "[" + counts[i] + "]");
		System.out.println();
	}
}
```

Now, to find the most popular color, we simply need to find out which of these color counts has the largest value (i.e., the maximum count). We will assume that there are no duplicates (or that only the first one with the maximum count is the answer):

```
	// Now determine the most popular color
	int	bestCountIndex = 0;
	for (int i=0; i<uniqueColors; i++) {
		if (counts[i] > counts[bestCountIndex])
			bestCountIndex = i;
	}
	System.out.println("The most popular color is: " + colors[bestCountIndex]);
```

5.5 Multi-Dimensional Arrays

JAVA allows arrays of multiple dimensions (2 or more). 2-dimensional (i.e., 2D) arrays are often used to represent data tables, game boards, mazes, pictures, terrains, etc…

In these cases, the information in the array is arranged by ***rows*** and ***columns***. Hence, to access or modify something in the table/grid, we need to specify which **row** AND **column** the item lies in. Therefore, instead of using just one index as with simple arrays, a 2D array requires that we always supply two indices … row and column.

Therefore, in JAVA, we specify a 2D array by using two sets of square brackets **[][]**. Therefore, our variables should have both sets of brackets when we declare them:

```
int[][]    schedule;  // a 2D array
int[]      list;      // a 1D array
int        age;       // not an array, just an int
```

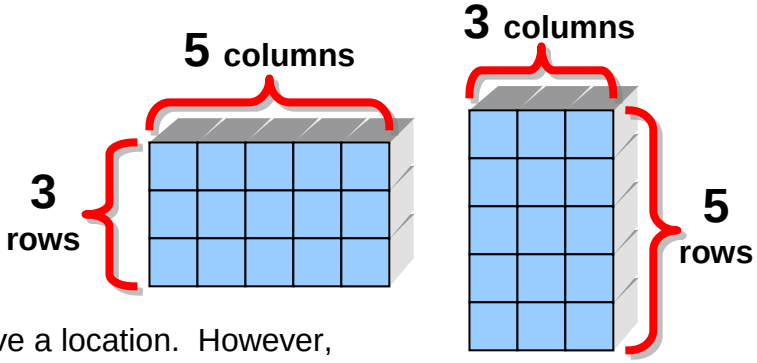
Also, when creating the arrays, we must specify the number of rows as well as the number of columns. Remember, the arrays cannot grow, so these should represent the maximum number of rows and columns that we want to have:

```
schedule = new int[10][10];      // table with 10 rows and 10 columns
image = new byte[1024][768];     // a 1024x768 pixel image
matchups = new String[12][4];    // 12x4 table of Strings
```

Usually, intuitively, the first length given represents the number of rows while the second represents the number of columns, but this need not be the case. For example, the following line of code creates an array that can hold 15 items:

```
grid1 = new int[3][5];
```

You can think of it as being either a **3x5** array or a **5x3** array. It is up to you as to whether the **3** is the rows or the **5** is the rows. You just need to make sure that you are consistent.

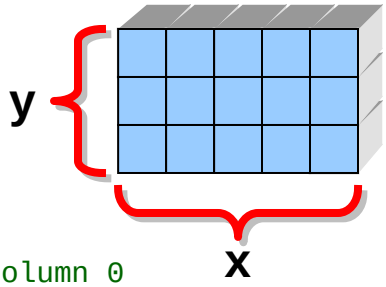


As with regular arrays, the elements each have a location. However, for 2D arrays, the location is a pair of indices instead of a single index. The rows and columns are all numbered starting with 0.

Therefore, we access and modify elements from the array by specifying both row and column indices as follows:

```
schedule[0][0] = 34;    // row 0, column 0
schedule[0][1] = 15;    // row 0, column 1
schedule[1][3] = 26;    // row 1, column 3
```

Sometimes, there is confusion, for example, when we create grids with **(x,y)** coordinates because when dealing with coordinates we always specify **x** before **y**. But visually, **x** represents the number of *columns* in a grid, while **y** represents the number of *rows* … hence **(x, y)** corresponds to **(columns, rows)** which seems counter-intuitive.




```
points[0][0] = 34;    // (x,y)=(0,0) = row 0, column 0
points[0][1] = 15;    // (x,y)=(0,1) = row 1, column 0
points[1][3] = 26;    // (x,y)=(1,3) = row 3, column 1
```

Example:

You are probably familiar with the game Sodoku. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The game begins with a starting board, where many of the spaces are not filled in (below left) and the user must complete the puzzle (below right):

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

How can we represent the above Sodoku boards with 2D arrays ? Well, it is really just a 2D array of numbers from 1 through 9 ... perhaps using 0 as an incomplete square.

Notice how we can represent the 2nd completed board by using the quick array declaration with the braces **{ }** as we did with 1D arrays:

```
byte[][]     board = {{5, 3, 4, 6, 7, 8, 9, 1, 2},
                      {6, 7, 2, 1, 9, 5, 3, 4, 8},
                      {1, 9, 8, 3, 4, 2, 5, 6, 7},
                      {8, 5, 9, 7, 6, 1, 4, 2, 3},
                      {4, 2, 6, 8, 5, 3, 7, 9, 1},
                      {7, 1, 3, 9, 2, 4, 8, 5, 6},
                      {9, 6, 1, 5, 3, 7, 2, 8, 4},
                      {2, 8, 7, 4, 1, 9, 6, 3, 5},
                      {3, 4, 5, 2, 8, 6, 1, 7, 9}};
```

Notice that there are more brace characters than with 1D arrays. Each row is specified by its own unique braces and each row is separated by a comma. In fact, each row is itself a 1D array. Interestingly, the **length** field of a multi-dimensional array returns the length of the first dimension only.

Consider this code:

```
int[][] ages = new int[4][7];
System.out.println(ages.length);      // displays 4, not 28!
```

In fact, we can actually access the separate arrays for each dimension:

```
int[][] ages = new int[4][7];
int[] firstArray = ages[0];          // gets 1st row from ages array
System.out.println(ages.length * firstArray.length);  // displays 28
```

Therefore, as you can see, a 2D array is actually an array of 1D arrays.

Iterating through a 2D array is similar to a 1D array except that we usually use 2 *nested* **for** loops. Here is some code to print out the board that we created above:

```
for (int i=0; i<board[0].length; i++) {
      for (int j=0; j<board.length; j++) {
            System.out.print(board[i][j]);
      }
      System.out.println();
}
```

This will produce the following output:

```
534678912
672195348
198342567
859761423
426853791
713924856
961537284
287419635
345286179
```

Of course, the output is not *pretty* because we cannot see the groupings into subgrids.

Here is a program that creates both boards with a pleasant output. It is a bit tricky to get the spacing right:

```
public class SodokuBoardDisplayProgram {

    public static void displayBoard(byte[][] aBoard) {
        for (int i=0; i<aBoard[0].length; i++) {
            if (i%3 ==0)
                System.out.println("+---+---+---+");
            for (int j=0; j<aBoard.length; j++) {
                if (j%3 ==0)
                    System.out.print('|');
                if (aBoard[i][j] == 0)
                    System.out.print(' ');
                else
                    System.out.print(aBoard[i][j]);
            }
            System.out.println('|');
        }
        System.out.println("+---+---+---+");
    }

    public static void main(String[] args) {
       byte[][]    board1 = {{5, 3, 0, 0, 7, 0, 0, 0, 0},
                             {6, 0, 0, 1, 9, 5, 0, 0, 0},
                             {0, 9, 8, 0, 0, 0, 0, 6, 0},
                             {8, 0, 0, 0, 6, 0, 0, 0, 3},
                             {4, 0, 0, 8, 0, 3, 0, 0, 1},
                             {7, 0, 0, 0, 2, 0, 0, 0, 6},
                             {0, 6, 0, 0, 0, 0, 2, 8, 0},
                             {0, 0, 0, 4, 1, 9, 0, 0, 5},
                             {0, 0, 0, 0, 8, 0, 0, 7, 9}};

      byte[][]     board2 = {{5, 3, 4, 6, 7, 8, 9, 1, 2},
                             {6, 7, 2, 1, 9, 5, 3, 4, 8},
                             {1, 9, 8, 3, 4, 2, 5, 6, 7},
                             {8, 5, 9, 7, 6, 1, 4, 2, 3},
                             {4, 2, 6, 8, 5, 3, 7, 9, 1},
                             {7, 1, 3, 9, 2, 4, 8, 5, 6},
                             {9, 6, 1, 5, 3, 7, 2, 8, 4},
                             {2, 8, 7, 4, 1, 9, 6, 3, 5},
                             {3, 4, 5, 2, 8, 6, 1, 7, 9}};

       System.out.println("Start:");
       displayBoard(board1);
       System.out.println("\n\nCompleted:");
       displayBoard(board2);
    }
}
```

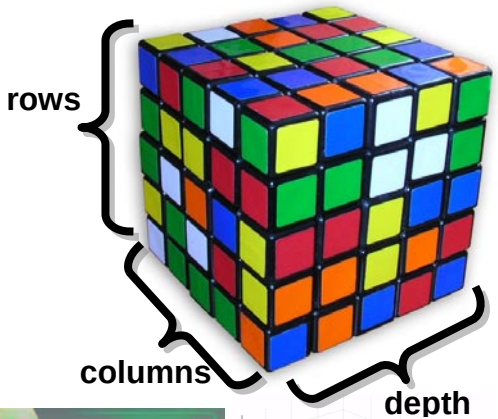
The code will produce the following more pleasant output:

```
Start:
+---+---+---+
|53 | 7 |   |
|6  |195|   |
| 98|   | 6 |
+---+---+---+
|8  | 6 |  3|
|4  |8 3|  1|
|7  | 2 |  6|
+---+---+---+
| 6 |   |28 |
|   |419|  5|
|   | 8 | 79|
+---+---+---+


Completed:
+---+---+---+
|534|678|912|
|672|195|348|
|198|342|567|
+---+---+---+
|859|761|423|
|426|853|791|
|713|924|856|
+---+---+---+
|961|537|284|
|287|419|635|
|345|286|179|
+---+---+---+
```

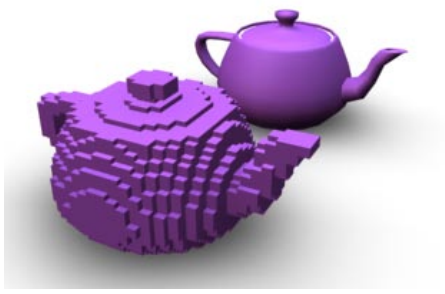
Interestingly, you can create even higher dimensional arrays. For example, an array as follows may be used to represent a cube of colored blocks:

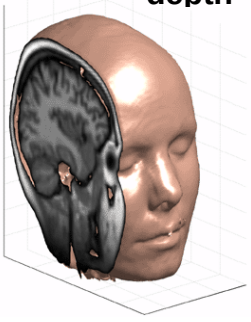
```
int[][][] cube = new String[5][5][5];
```



Notice that there are now 3 sets of square brackets. Using 3D arrays works the same way as with 2D arrays except that we now use 3 sets of brackets and 3 indices when referring to the elements of the array.

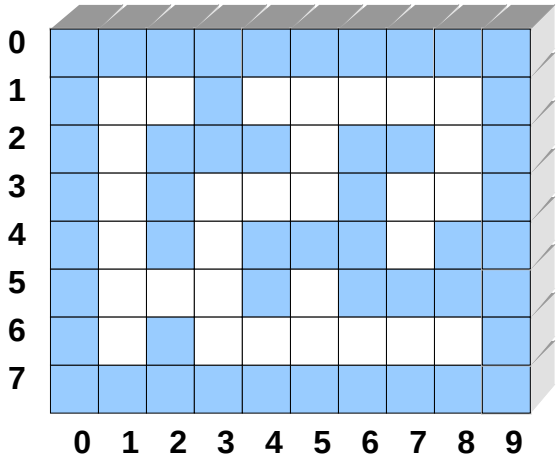
3-dimensional arrays are often used in the real world to model various objects:





Example:

Consider representing the following maze by using arrays. We could represent this maze by using a 2D array of **bytes** indicating whether or not there is a wall at each location in the array (i.e., **1** for wall, **0** for open space).



We can create the maze in a similar manner to our Sodoku board by making a quick array declaration with the braces **{ }**:

```
byte[][]     maze = {{1,1,1,1,1,1,1,1,1,1},
                     {1,0,0,1,0,0,0,0,0,1},
                     {1,0,1,1,1,0,1,1,0,1},
                     {1,0,1,0,0,0,1,0,0,1},
                     {1,0,1,0,1,1,1,0,1,1},
                     {1,0,0,0,1,0,1,1,1,1},
                     {1,0,1,0,0,0,0,0,0,1},
                     {1,1,1,1,1,1,1,1,1,1}};
```

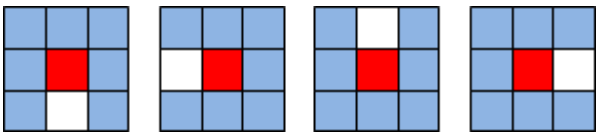
This array represents a grid with 8 rows and 10 columns. We could display this array quite simply by iterating through the rows and columns:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (maze[row][col] == 1)
          System.out.print ('*');
        else
          System.out.print (' ');
    }
    System.out.println();
}
```

```
**********
*  *     *
* *** ** *
* *   *  *
* * *** **
*   * ****
* *      *
**********
```

Some mazes contain “dead-ends”. A “dead-end” is any location in the maze that has only one way to get into it (i.e., it is surrounded by 3 walls). There are 6 dead-ends in our maze above. How can we adjust the code so that dead-ends are indicated with a '@' character ?

We need to identify a dead-end by checking the grid locations around it. There are exactly 4 cases that we should check as shown here →



Given a particular grid location, we can determine if it is a dead end by checking the locations around it as follows:

```
if (the location is an open space) {
        if ((the locations left AND above AND right are all walls) OR
           (the locations above AND right AND below are all walls) OR
           (the locations left AND right AND below are all walls) OR
           (the locations left AND above AND below are all walls)) {
                mark this location as a dead end
        }
}
```

The logic follows the 4 cases shown in the diagram. However, how do we write this using the **maze** array ? Well, let us assume that we are looking at the location in the **maze** at a specific row **r** and column **c**. We need to access the array to check each location left, right, above and below by varying the row and column with respect to **r** and **c**. Also, we need to decide how to "mark" the location as a dead end. Perhaps we can place a value of **2** there. Here is the code:

```
if (maze[r][c] == 0) {
    if (((maze[r][c-1] == 1) && (maze[r-1][c] == 1) && (maze[r][c+1] == 1)) ||
        ((maze[r-1][c] == 1) && (maze[r][c+1] == 1) && (maze[r+1][c] == 1)) ||
        ((maze[r][c-1] == 1) && (maze[r][c+1] == 1) && (maze[r+1][c] == 1)) ||
        ((maze[r][c-1] == 1) && (maze[r-1][c] == 1) && (maze[r+1][c] == 1))) {

          maze[r][c] = 2;
    }
}
```

Notice that the **AND** and **OR** operators from our pseudocode are replaced with the Java operators **&&** and **||**.

The code looks a little long and there is quite a bit of potential for making mistakes with the +1, -1 array indexing. Can we do this in a simpler way, given that all maze locations are represented as numbers either 0 or 1 ?

Yes. You may have noticed that the dead-end locations have exactly 3 walls around them. If we were to add up the values in the maze locations in the 4 directions around (**r**,**c**), for *dead-ends* therefore, we should obtain a count of 3. All other *non-dead-end* locations will have values of 0, 1, 2 or 4. Therefore, we can simplify the code as follows:

```
if (maze[r][c] == 0) {
    int count = maze[r][c-1] + maze[r][c+1] + maze[r-1][c] + maze[r+1][c];
    if (count == 3)
          maze[r][c] = 2;
}
```

I am sure that you will agree that this code is simpler and still easy to understand. So, then, we can write a procedure called **`markDeadEnds(byte[][] aMaze)`** that takes the maze and marks the dead ends as follows:

```
public static void markDeadEnds(byte[][]  m) {
   for (int r=0; r<m.length; r++) {
      for (int c=0; c<m[0].length; c++) {  // m[0] is the first row
         if (m[r][c] == 0) {
            int count = m[r][c-1] + m[r][c+1] + m[r-1][c] + m[r+1][c];
            if (count == 3)
               m[r][c] = 2;
         }
      }
   }
}
```

What do you think of the above code ? Is it simple and logical ? Are there any problems ?

There could potentially be a problem with the boundaries. Our code assumes that **(r,c)** is not along the border, otherwise we may be trying to examine locations that are outside of the maze boundaries. For example, consider the first values of **r** and **c** ... which are both zeroes. This represents the top left corner wall of the maze. Our code attempts to count the values at `m[r][c-1]`, `m[r][c+1]`, `m[r-1][c]`, and `m[r+1][c]`. For the case when r=0 and c=0, the values of c-1 and r-1 are -1. So we will be trying to access the array maze[0][-1] and maze[-1][0].

Fortunately, our maze was constructed with walls all around the outside, so this condition will never occur because **m[r][c]** will never be **0** on the border locations ... so the **if** statement ensures that we never try to access outside of the array.

If however, the maze had *openings* along the border, we would have to check for such boundary issues by adjusting our FOR loops so that the first and last rows and columns are ignored:

```
for (int r=1; r<m.length-1; r++) {
   for (int c=1; c<m[0].length-1; c++) {  // m[0] is the first row
         // ...
   }
}
```

We will need to add something to our display routine so that the @ symbols is displayed at a dead-end:

```
for (int row=0; row<8; row++) {
    for (int col=0; col<10; col++) {
        if (maze[row][col] == 2)        // Checks for dead-ends
            System.out.print ('@');
        else if (maze[row][col] == 1)
            System.out.print ('*');
        else
            System.out.print (' ');
    }
    System.out.println();
}
```

Now, if we run the code ... here is the result ...

```
**********
* @*@@   *
* *** ** *
* *    *  *
* * ***@**
*   *@****
*@*  @  @*
**********
```

There is a problem. It is showing 8 dead-ends instead of 6. Two of them are wrong. What happened ? Why are these extra dead-ends showing up as dead-ends ?

Well, take a look at the topmost wrong one. There is a wall above ... nothing to the right or below ... and a dead-end to the left. So ... 1 for the wall and 2 for the dead-end adds to 3. That is why the dead-end is showing up ... because the total count around it is 3, even though there is only one wall.

The problem is a result of our code simplification and our choice to use the number 2 as a dead-end value. Perhaps we should choose a larger value (above 3) so that the total count will never be 3. We can make this change in our code:

```
if (count == 3)
   m[r][c] = 5;
```

And of course, alter our display routine as well by displaying @ when 5 is there, not 2:

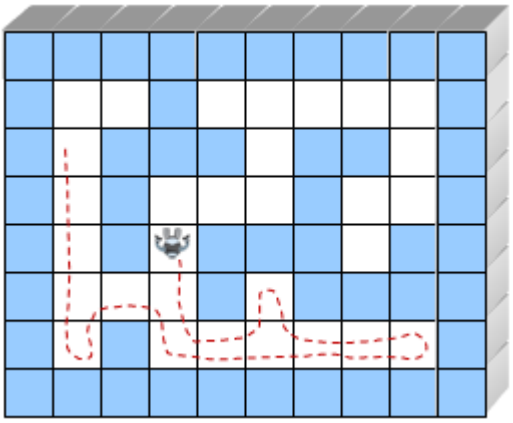
```
if (maze[row][col] == 5)        // Checks for dead-ends
   System.out.print ('@');
```

When we run the code now, we obtain the proper result:

```
**********
* @*@    *
* *** ** *
* *    *  *
* * ***@**
*   *@****
*@*     @*
**********
```

The completed code can be found online in the class **MazeTestProgram.java.**

Example:



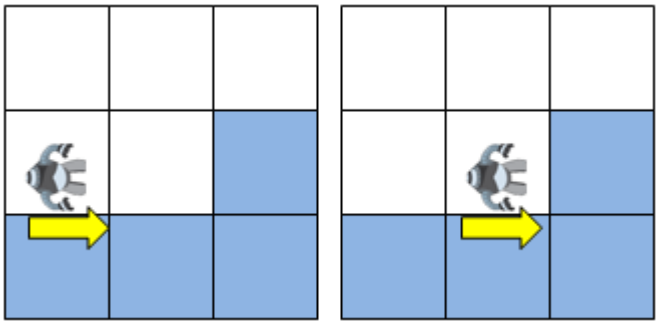
How can we write code to simulate a robot travelling through the maze ? Assume that we simply want the robot to continuously travel around the maze without stopping. How can we do this ?

One well-known method of traveling through a maze is that of using the “right-hand rule” ... which says that as long as we keep our right hand touching a wall as we walk, we will find a solution through the maze.

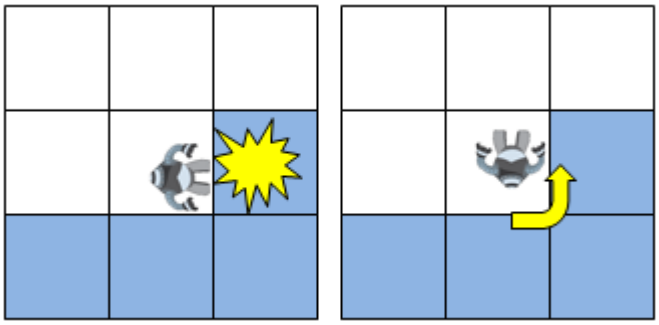
Does this work for all mazes ? I guess it depends on what we call a “maze”.

Mazes with *inner loops* cannot necessarily be solved using this strategy … it would depend on where the robot began following the walls. We will assume, however, that we have mazes with no inner loops.

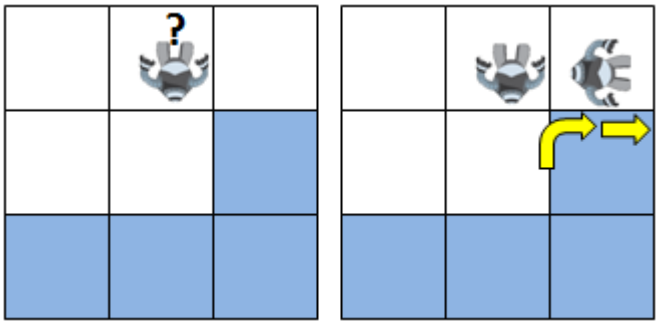
Let us consider how the robot will move around in the maze to follow the “right-hand rule”. Assume that the robot has its right "hand" on a wall. If it is able to move forward, it should simply do so as shown here →



If however, the robot encounters a wall in front of it, then it should turn left →

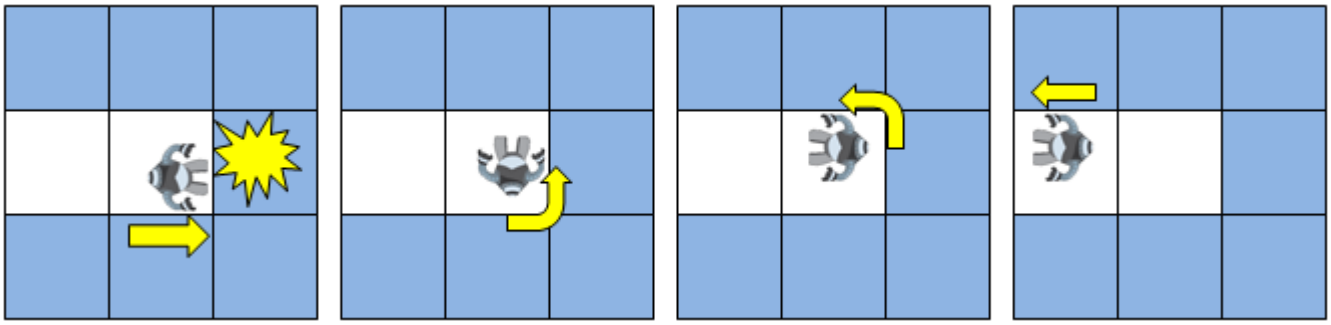


Then the robot should continue moving forwards as it now has its right hand on the wall again. It is possible that while traveling along it may lose contact with the wall →

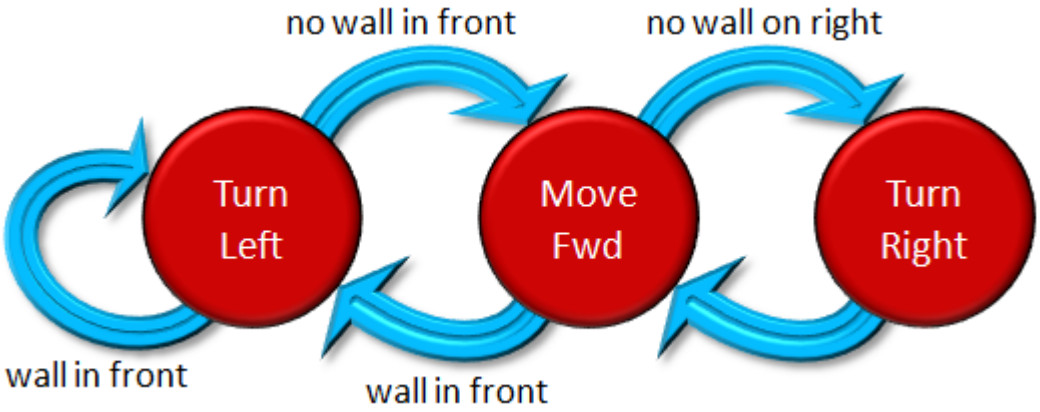


In this case, notice that the robot needs to regain contact again with the wall on its right. First, it must turn right. However, the wall will still not be on the right of the robot after turning. Therefore it also needs to move forward.

Then, it will be back-on-track again. These three cases actually encompass all situations. For example, if the robot ends up in a "dead-end", these three cases will get it out again.



So, how can we write an algorithm for traveling through this maze based on our understanding of the right-hand rule ? We simply follow the wall-following model that we developed and produce a ***state diagram***. Basically, the robot can be in one of 3 ***states*** at all times ... moving **forward**, turning **left** or turning **right** (we will assume that the robot never stops). A state diagram will indicate when (i.e., what condition must occur) we need to switch from one state to another. Notice here how the various conditions cause changes to the three states:



Here is the algorithm, which assumes that each move operation moves the robot one square:

```
repeat {
        if (there is a wall on the right) then {
                if (there is a wall in front) then
                        turnLeft()
                otherwise
                        moveForward()
        }
        otherwise {
                turnRight()
                moveForward()
        }
}
```

The code is straight forward. However, some details have been left out. For example, how do we know if there is a wall on the right or not ?

It depends on the robot's current location in the maze as well as its current direction. So, we need to know at all times which direction the robot is facing and which maze location it is in. The values for the row and column will have to be valid locations in the maze. Assume that the maze has either 1 or 0 at each (row, column) location representing walls or open spaces, respectively:

```
public static byte[][]     maze = {{1,1,1,1,1,1,1,1,1,1},
                                   {1,0,0,1,0,0,0,0,0,1},
                                   {1,0,1,1,1,0,1,1,0,1},
                                   {1,0,1,0,0,0,1,0,0,1},
                                   {1,0,1,0,1,1,1,0,1,1},
                                   {1,0,0,0,1,0,1,1,1,1},
                                   {1,0,1,0,0,0,0,0,0,1},
                                   {1,1,1,1,1,1,1,1,1,1}};
```

We could then perhaps start the robot in the top left corner at row 1, column 1 (i.e., (1,1) as shown red in the array above). So, we can store the robot's location as follows:

```
public static int robotRow = 1;
public static int robotCol = 1;
```

The direction of the robot will need to be stored as well. We can define a unique constant for each of the 4 directions as follows:

```
public static final byte EAST = 0;
public static final byte NORTH = 1;
public static final byte WEST = 2;
public static final byte SOUTH = 3;
```

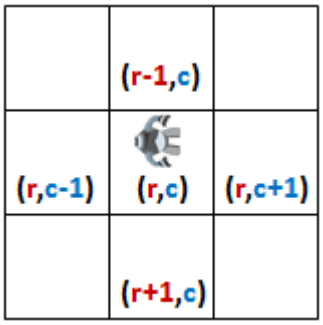
Then to start, we could set the robot's direction to SOUTH so that it has its hand on the outside wall and is facing downwards.

```
public static int direction = SOUTH;
```

How do we determine whether or not there is a “wall on the right” of the robot ? We would need to look at the position to its right in the maze. Let's write a function to do this. We will need to consider the maze, the robot's current location and its direction. Then the method could return **true** or **false** indicating whether or not there is a wall on the right of the robot:

```
public static boolean wallOnRight() {
      // ...
}
```

Assuming that the robot is at position **(r, c)**, then the locations around the robot are determined as shown here →



To complete the **wallOnRight()** function, we will need to consider all 4 directions separately. The code follows from the diagram here. We simply consider each direction and look in the maze on the right hand side of the robot to see if there is a wall there:

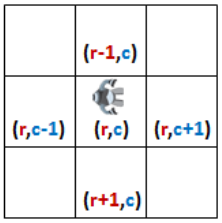
```
if (((d is EAST) AND (maze[r+1][c] is a wall)) OR
    ((d is NORTH) AND (maze[r][c+1] is a wall)) OR
    ((d is WEST) AND (maze[r-1][c] is a wall)) OR
    ((d is SOUTH) AND (maze[r][c-1] is a wall))) then {
        return true;
else
        return false;
```

This code can be simplified by using a **switch** statement as follows:

```
public static boolean wallOnRight() {
   switch(direction) {
      case EAST:  return (maze[robotRow+1][robotCol] == 1);
      case NORTH: return (maze[robotRow][robotCol+1] == 1);
      case WEST:  return (maze[robotRow-1][robotCol] == 1);
      case SOUTH: return (maze[robotRow][robotCol-1] == 1);
   }
   return false;
}
```

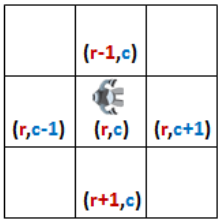
A similar check can be made for walls in front of the robot. We just need to adjust the directions in our above code. Here is a function that checks if a wall lies ahead:

```
public static boolean wallAhead() {
   switch(direction) {
      case SOUTH: return (maze[robotRow+1][robotCol] == 1);
      case EAST:  return (maze[robotRow][robotCol+1] == 1);
      case NORTH: return (maze[robotRow-1][robotCol] == 1);
      case WEST:  return (maze[robotRow][robotCol-1] == 1);
   }
   return false;
}
```



To move the robot forward, we need to make a similar check so that we alter the row and column of the robot based on the current direction. Here is the code. I believe that it should be straight forward:

```
public static void moveForward() {
      switch(direction) {
            case SOUTH: robotRow = robotRow + 1; return;
            case EAST:  robotCol = robotCol + 1; return;
            case NORTH: robotRow = robotRow - 1; return;
            case WEST:  robotCol = robotCol - 1; return;
      }
}
```



Now, how do we make a turn ? Turning does not change the location of the robot, it just changes the direction. Let's assume that each turn makes a 90 degree change in direction (e.g., from EAST to NORTH or from EAST to SOUTH, but never from EAST to WEST).

Recall that our directions are simply numbers:

```
public static final byte EAST = 0;
public static final byte NORTH = 1;
public static final byte WEST = 2;
public static final byte SOUTH = 3;
```

Look at the order of these direction numbers. Hopefully you can see that they were chosen carefully so that increasing one of these numbers by 1 represents a left turn and decreasing the number by 1 results in a right turn. So then, turning left or right is as simple as:

```
direction = direction + 1;    // left turn
direction = direction - 1;    // right turn
```

The only issue with the above formula is when we want to go from SOUTH to EAST or from EAST to SOUTH. The formula would change the direction to 4 and -1 in these two cases ... which are invalid directions.

We can add two IF statements to handle this as follows:

```
if (direction == -1)
      direction = 3;
if (direction == 4)
      direction = 0;
```

But we can improve this. Whenever we have such a situation, where we are increasing/decreasing a value and need to have it ***wrap around*** to the beginning/end again, the modulo operator becomes quite useful. You may recall that the modulo operator returns the remainder after dividing by some number. So whenever we need to count from 0 to some number N, if we always let our counter be "modulo N", then it will always count from 0 to N-1.

For example, consider a counter **d** which goes from 0 to 15. Notice the result when we determine the value of **d** modulo 4 (which is **d**%4 in JAVA).

```
d          0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15
d%4        0   1   2   3   0   1   2   3   0   1   2   3   0   1   2   3
(d+1)%4    1   2   3   0   1   2   3   0   1   2   3   0   1   2   3   0
```

In this case, the modulo operator simply discards multiples of 4 from the counter and gives us the remainder each time. This is exactly what we want to do! Notice how the 4 becomes 0. We can therefore do the following for turning left:

```
direction = (direction + 1) % 4;
```

Notice how it will work for our left direction:

```
d          0   1   2   3
(d+1)%4    1   2   3   0     // Left Turn
```

But now look for the right direction. We need to add 4 to avoid the -1:

```
d           0   1   2   3
(d-1)%4    -1   0   1   2    // Right Turn ... not quite right
(d-1+4)%4   3   0   1   2    // Right Turn ... correct now
```

So turning right will be as follows:

```
direction = (direction + 3) % 4;
```

Now we can merge all of this code together to come up with a complete solution. Here is the resulting JAVA code which makes use of the methods and turning formulas that we just finished discussing. As a reminder, the pseudocode for travelling the maze is on the left:

```
repeat {
    if (there is a wall on the right) then {
        if (there is a wall in front) then
            turnLeft()
        otherwise
            moveForward()
    }
    otherwise {
        turnRight()
        moveForward()
    }
}
```

```
while (true) {
    if (wallOnRight()) {
        if (wallAhead())
            direction = (direction + 1) % 4;
        else
            moveForward();
    }
    else {
        direction = (direction + 3) % 4;
        moveForward();
    }
}
```

It sure is easy to write once we have the pseudocode! You should always get used to writing pseudocode beforehand to make sure that your logic is correct. Here is the final code. Notice that we added an extra **IF** statement in the **displayMaze()** procedure so that it draws the robot as an **'R'**. The main method runs in an infinite loop. If you do not like this, you can create a counter so that the robot only moves a specific number of times.

```
public class MazeTravelProgram {

    public static final byte EAST = 0;
    public static final byte NORTH = 1;
    public static final byte WEST = 2;
    public static final byte SOUTH = 3;

    public static byte[][]   maze = {{1,1,1,1,1,1,1,1,1,1},
                                     {1,0,0,1,0,0,0,0,0,1},
                                     {1,0,1,1,1,0,1,1,0,1},
                                     {1,0,1,0,0,0,1,0,0,1},
                                     {1,0,1,0,1,1,1,0,1,1},
                                     {1,0,0,0,1,0,1,1,1,1},
                                     {1,0,1,0,0,0,0,0,0,1},
                                     {1,1,1,1,1,1,1,1,1,1}};
    public static int robotRow = 1;
    public static int robotCol = 1;
    public static int direction = SOUTH;

    public static void displayMaze(byte[][] m) {
        for (int row=0; row<8; row++) {
            for (int col=0; col<10; col++) {
                if ((row == robotRow) && (col == robotCol))
                    System.out.print ('R');
                else {
                    if (m[row][col] == 5)
                        System.out.print ('@');
```

```
                    else if (m[row][col] == 1)
                        System.out.print ('*');
                    else
                        System.out.print (' ');
                }
            }
            System.out.println();
        }
    }

    public static boolean wallOnRight() {
        switch(direction) {
            case EAST:  return (maze[robotRow+1][robotCol] == 1);
            case NORTH: return (maze[robotRow][robotCol+1] == 1);
            case WEST:  return (maze[robotRow-1][robotCol] == 1);
            case SOUTH: return (maze[robotRow][robotCol-1] == 1);
        }
        return false;
    }

    public static boolean wallAhead() {
        switch(direction) {
            case SOUTH: return (maze[robotRow+1][robotCol] == 1);
            case EAST:  return (maze[robotRow][robotCol+1] == 1);
            case NORTH: return (maze[robotRow-1][robotCol] == 1);
            case WEST:  return (maze[robotRow][robotCol-1] == 1);
        }
        return false;
    }

    public static void moveForward() {
        switch(direction) {
            case SOUTH: robotRow = robotRow + 1; return;
            case EAST:  robotCol = robotCol + 1; return;
            case NORTH: robotRow = robotRow - 1; return;
            case WEST:  robotCol = robotCol - 1; return;
        }
    }

    public static void main(String[] args) {
        displayMaze(maze);
        while (true) {
            if (wallOnRight()) {
                if (wallAhead())
                    direction = (direction + 1) % 4;
                else
                    moveForward();
            }
            else {
                direction = (direction + 3) % 4;
                moveForward();
            }
            displayMaze(maze);
        }
    }
}
```

Here are the first 20 maze printouts of the running code, showing the robot location in red and the wall on its right in blue: