Chapter 6

Loops and ArrayLists

What is in this Chapter?

When programming, it is often necessary to repeat a selected portion of code a specific number of times, or until some condition occurs. We will look here at the **FOR** and **WHILE** loop constructs that are available in most programming languages. It is also common, when programming, to gather (or collect) a bunch of data (or objects) together. This is commonly referred to as a **collection**. We will look at the simplest way of collecting information together by using the **ArrayList**.
6.1 Repeating Code Using For and While Loops

Suppose that you wanted to ask the user for 5 exam marks and then print the average exam mark. You might write a program that looks like this:

```java
int n1, n2, n3, n4, n5;

System.out.print("Enter exam mark 1:  ");
n1 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 2:  ");
n2 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 3:  ");
n3 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 4:  ");
n4 = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 5:  ");
n5 = new Scanner(System.in).nextInt();

System.out.println("The average is " + ((n1+n2+n3+n4+n5)/5));
```

The above code gets all the exam marks first and stores them into variables... afterwards computing the sum and average. Instead of storing each number separately, we can add each number to an accumulating `sum` as follows:

```java
int sum;

System.out.print("Enter exam mark 1:  ");
sum = new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 2:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 3:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 4:  ");
sum = sum + new Scanner(System.in).nextInt();
System.out.print("Enter exam mark 5:  ");
sum = sum + new Scanner(System.in).nextInt();

System.out.println("The average is " + (sum/5));
```

While this code may work fine, what would happen if we needed 100 numbers? Clearly, some part of the code is repeating over and over again (i.e., adding the next number to the `sum`).

What if we could somehow tell JAVA to repeat something over and over again like this:
sum = 0  
REPEAT 100 TIMES {  
    Get the next exam mark and add it to the sum  
}  

Well, in JAVA, we do have a way of repeating code using something called a **FOR loop**. Here is how we could modify our program to get 100 numbers:

```java
int sum = 0;
for (int count=1; count<=100; count++) {
    System.out.print("Enter exam mark "+ count + ": ");
    sum = sum + new Scanner(System.in).nextInt();
}
System.out.println("The average is "+ (sum/100));
```

Notice that the for loop has brackets () and then is followed by braces {} which contains the body of the loop (i.e., the code that is to be repeated). Let's take a look at what is inside the () brackets.

Notice that it declares an `int` variable called `count` and gives it a value of 1 to begin. This variable `count` is called a **loop variable** and in our example it represents a **counter**. After the first semi-colon ; there is a conditional JAVA expression `count<=100` ... which is called the **loop stopping condition**. That is, the loop will keep repeating as long as our counter is less than or equal to 100. The moment the `count` reaches 101, the loop stops and our answer is printed. After one more semi-colon ; there is the **loop update expression** code `count++` which is evaluated each time the loop completes a cycle. In our case, each time through the loop we just need to increase the counter by 1.

Notice that we can even use the value of the `count` variable within the loop. In our case we are printing out the counter in order to indicate which exam mark we are going to enter. In general, a loop variable can be used any time within the loop but it cannot be used outside the loop body.

In our example, what if we did not know how many time to repeat (i.e., we don’t know how many exam marks there will be) ? Well, in that case we can ask the user of the program for the total number as follows …
import java.util.Scanner;

class CalculateAverageProgram {
    public static void main(String args[]) {
        int nums, sum;
        System.out.println("How many exam marks do you want to average ?");
        nums = new Scanner(System.in).nextInt();
        sum = 0;

        // Get the numbers one at a time, and add them
        for (int count=1; count<=nums; count++) {
            System.out.print("Enter exam mark "+count + ": ");
            sum += new Scanner(System.in).nextInt();
        }
        System.out.println("The average is " + (sum / nums));
    }
}

Notice that the program is now flexible in the number of exam marks that it is able to average. Here is an example of the output:

How many exam marks do you want to average ?
5
Enter exam mark 1: 10
Enter exam mark 2: 67
Enter exam mark 3: 43
Enter exam mark 4: 96
Enter exam mark 5: 20
The average is 47

Here is another example. Suppose that we wanted to print out the odd numbers from 1 to 100. How could we do this? Do you know how to check whether or not a number is odd?

We can check if the remainder after dividing by two is zero. The modulus operator % gives the remainder after dividing, so we do n%2 on number n. We just need to put this into a for loop:

for (int n=1; n<=100; n++) {
    if ((n%2) > 0) {
        System.out.println(n);
    }
}

Notice that we can use the same counter (but called it n this time). Then in the loop we just check if the modulus is non-zero and print out the number in that case since it would be odd.

We could eliminate the if statement by simply counting by twos (starting at 1) as follows …
for (int n=1; n<=100; n=n+2) {
    System.out.println(n);
}

The above code too will print out only the odd numbers from 1 to 100 since now the counter \( n \) increases by two each time, thereby skipping over all the even numbers. You should realize that the update expression can be any JAVA code.

Here is another example that prints out all the even numbers backwards from 100 to 1:

for (int n=100; n>0; n=n-2) {
    System.out.println(n);
}

Notice how we started \( n \) at a higher value now and that we subtract by two each time (i.e., 100, 98, 96, etc.). Also, notice that the stopping condition now uses a \( > \) instead of \( \leq \) (i.e., as long as \( n \) is above zero we keep decreasing it by 2).

What would happen if the stopping-expression evaluated to false right away as follows:

for (int n=1; n>=100; n++) {
    System.out.println(n);
}

In this case, \( n \) starts at 1 and the stopping condition determines that it is not greater than or equal to 100. Thus, the loop body never gets evaluated. That is, the for loop does nothing ... your program ignores it.

A similar unintentional situation may occur if you accidentally place a semi-colon ; after the round brackets by mistake:

for (int n=1; n<=100; n++); {
    System.out.println(n);
}

In this situation, JAVA assumes that the for loop ends at that semi-colon ; and that it has no body to be evaluated. In this case, the body of the loop is considered to be regular code outside of the for loop and it is evaluated once. Hence JAVA "sees" the above code as:

for (int n=1; n<=100; n++){
} System.out.println(n);
One last point regarding for loops is that you do not need the braces around the loop body if the loop body contains just one JAVA expression (i.e., just like the if statement):

```java
for (int n=100; n>0; n=n-2)
    System.out.println(n);
```

In that case though, you should still indent your code so that it is clear what is in the loop.

**WHILE LOOPS:**

In some situations, we do not know how many times to repeat something. That is, we may need to repeat some code until a particular condition occurs. For example, we may wish to enter exam marks and find their average but we may not know how many exams we have. Instead of forcing the user to count them all ahead of time, we can allow him to enter the marks one at a time and then enter a "special" value that will indicate the completion of the entries (e.g., -1).

Whenever we have such a situation, we use the easier while loop which has this format:

```java
while (loop stopping condition) {
    ...
}
```

Here is how it can be used in our exam mark program:

```java
import java.util.Scanner;

class CalculatorAverageProgram2 {
    public static void main(String args[]) {
        int count, sum, num;
        count = 1;
        sum = 0;
        num = 0;

        while (num >= 0) {
            System.out.print("Enter exam mark "+ count + " (use -1 to quit): ");
            num = new Scanner(System.in).nextInt();
            if (num >= 0) {
                sum += num;
                count++;
            }
        }
        System.out.println("The average is "+ (sum / (count-1)));
    }
}
```
The stopping condition must evaluate to true or false ... and in this case it starts off with true since num=0. When the user enters -1, the stopping condition will become false and the loop will end.

Just as with for loops, you should be careful not to put a semi-colon ; after the round brackets, otherwise your loop body will not be evaluated. Usually your code will loop forever because the stopping condition will likely never change to false:

```java
while (num < 100) {
    System.out.println(n++); // This code will loop forever
}
```

As with the if statements and for loops, the braces { } are not necessary when the loop body contains a single JAVA expression:

```java
while (num >= 0)
    System.out.println(n++);
```

Some students tend to confuse the while loop with if statements and try to replace an if statement with a while loop. Do you understand the difference in the two pieces of code below?

```java
if (aPerson.age > 18)
    discount = 0;     discount = 0;
while (aPerson.age > 18)
    discount = 0;
```

Assume that the person’s age is 20. The leftmost code will set the discount to 0 and move on. The rightmost code will loop forever, continually setting the discount to 0.

### 6.2 Collecting Objects Together Using ArrayLists

In real life, objects often appear in groups. For example, ParkingLots contain multiple Cars, Banks contain multiple BankAccounts, Stores have multiple Customers, a Student has multiple Assignment marks etc..

When programming, we often need to group together objects in order to treat the group itself as a kind of “container” object that contains other objects. In JAVA there are many kinds of container-type objects. We will look at the most common one called an ArrayList.
An **ArrayList** is an object that contains multiple arbitrary objects. To create an **ArrayList**, we can simply call the constructor from JAVA's **ArrayList** class. Here is an example of creating an **ArrayList** and storing it in a variable so that we can use it:

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

This is the general format for an **ArrayList** that can hold any kinds of objects. However, it is "highly recommended" that we specify the type of objects that will be stored in the **ArrayList**. We do this by specifying the type between `<` and `>` characters just before the round brackets (`()`) as follows:

```java
ArrayList<Object> myList;
myList = new ArrayList<Object>();
```

The above code allows us to store any kind of object in the **ArrayList**. We then use the **ArrayList**'s `add()` method to add an object to the list as follows:

```java
import java.util.ArrayList;

class ArrayListTestProgram {
    public static void main(String args[]) {
        ArrayList<Object> myList;

        myList = new ArrayList<Object>();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Truck());
        System.out.println(myList);
    }
}
```

It is as if the **ArrayList** is a kind of “bag” into which we can place any objects. The **ArrayList** keeps the objects altogether so that we can pass the list around now as a single object … just like we can consider a "bag" to be a single object, although it may contain many other objects.

Notice in the above code that we are adding a **String**, an **int**, a **Person** object and a **Truck** object. Notice as well that at the top of the program we imported `java.util.ArrayList`. This is necessary in order for JAVA to know where to find the **ArrayList** class and its methods. The output for the program is as follows (assuming that **Person** and **Truck** do not have `toString()` methods):

```
Hello, 25, Person@addbf1, Truck@42e816
```
Did you notice how ArrayLists look when you print them out? They show all the elements/items in the list separated by commas, in between square brackets[].

Often, the objects in an ArrayList are of the same type, but this need not be the case. If we know, for example, that all of the objects in the ArrayList will be Strings (e.g., names of people), then we should declare and create the list as follows:

```java
ArrayList<String> myList;
myList = new ArrayList<String>();
...
```

Similarly, if the objects to be stored in the list were of type Person, BankAccount or Car… then we would specify the type as <Person>, <BankAccount> or <Car>, respectively.

In JAVA, ArrayLists keep their items stored in the order that we added them and it assigns each item its own unique index (i.e., a number that is used to identify its position in the list). The index of the first item in the list is always 0.

The ArrayList class has a method called `get()` that allows us to extract the item at a given position in the list. We supply the desired position as a parameter to the method. If we want the first item, we use position 0 by saying `get(0)` to the ArrayList.

Another useful ArrayList method is the `size()` method. Which returns the number of elements currently in the list.

Here is an example that uses these methods:

```java
ArrayList<Object> myList;
myList = new ArrayList<Object>();
System.out.println(myList.size()); // outputs 0
myList.add("Hello");
myList.add(25);
myList.add(new Person());
myList.add(new Car());
System.out.println(myList.get(0)); // outputs "Hello"
System.out.println(myList.get(2)); // outputs Person@addbf1
System.out.println(myList.get(4)); // an IndexOutOfBoundsException
System.out.println(myList.size()); // outputs 4
```
6.3 Team/League Example

Let's consider a realistic use of the ArrayList object by creating classes called Team and League in which a League object will contain a bunch of Team objects. That is, the League object will have an instance variable of type ArrayList to hold onto the multiple Team objects within the league.

Consider first the creation of a Team class that will represent a single team in the league. For each team, we will maintain the team's name as well as the number of wins, losses and ties for the games that they played. Here is the basic class (review the previous chapters in the notes if any of this is not clear):

```java
class Team {
    String name; // The name of the Team
    int wins; // The number of games that the Team won
    int losses; // The number of games that the Team lost
    int ties; // The number of games that the Team tied

    Team(String aName) {
        this.name = aName;
        this.wins = 0;
        this.losses = 0;
        this.ties = 0;
    }

    public String toString() {
        return ("The " + this.name + " have " + this.wins + " wins, " +
                this.losses + " losses and " + this.ties + " ties.");
    }

    // Returns the total number of points for the team
    int totalPoints() {
        return (this.wins * 2 + this.ties);
    }

    // Returns the total number of games played by the team
    int gamesPlayed() {
        return (this.wins + this.losses + this.ties);
    }
}
```
We can test out our Team object with the following test code, just to make sure it works:

```java
class TeamTestProgram {
    public static void main(String args[]) {
        Team teamA, teamB;

        teamA = new Team("Ottawa Senators");
        teamB = new Team("Montreal Canadians");

        // Simulate the playing of a game in which teamA beat teamB
        System.out.println(teamA.name + " just beat " + teamB.name);
        teamA.wins++;
        teamB.losses++;

        // Simulate the playing of another game in which they tied
        System.out.println(teamA.name + " just tied " + teamB.name);
        teamA.ties++;
        teamB.ties++;

        // Now print out some statistics
        System.out.println(teamA);
        System.out.println(teamB);
        System.out.print("The " + teamA.name + " have ");
        System.out.println(teamA.totalPoints() + " points and played ");
        System.out.println(teamA.gamesPlayed() + " games.");
        System.out.print("The " + teamB.name + " have ");
        System.out.println(teamB.totalPoints() + " points and played ");
        System.out.println(teamB.gamesPlayed() + " games.");
    }
}
```

Here is what the Team objects look like after playing the two games:
Here is the output from our little test program:

Ottawa Senators just beat Montreal Canadians
Ottawa Senators just tied Montreal Canadians
The Ottawa Senators have 1 wins, 0 losses and 1 ties.
The Montreal Canadians have 0 wins, 1 losses and 1 ties.
The Ottawa Senators have 3 points and played 2 games.
The Montreal Canadians have 1 points and played 2 games.

Now let us implement the **League** class. A league will also have a **name** as well as an **ArrayList** (called **teams**) of **Team** objects. Here is the basic class structure (notice the **import** statement at the top):

```java
import java.util.ArrayList;

class League {
    String name;
    ArrayList<Team> teams;

    League(String n) {
        this.name = n;
        this.teams = new ArrayList<Team>(); // Doesn’t make any Team objects
    }

    // This specifies the appearance of the League
    public String toString() {
        return "The " + this.name + " league";
    }
}
```

Notice that the **ArrayList** is created within the constructor and that it is initially empty. That means, a brand new league has no teams in it. It is important to note also that there are no **Team** objects created at this time.

At this point, we have defined two objects: **Team** and **League**. One thing that we will need to do is to be able to add teams to the league. Here is an example of how we can create a league with three teams in it:

```java
League nhl;

nhl = new League("NHL");
nhl.teams.add(new Team("Ottawa Senators")(nhl.teams.add(new Team("Montreal Canadians")));
nhl.teams.add(new Team("Toronto Maple Leafs"));
```

In order to add the team to the league, we simply add it to the league's **teams** by using the **add()** method that is already defined in the **ArrayList** class. Here is a diagram showing how the **League** object stores the 3 **Teams** ...
Suppose now that we wanted to print out the teams in the league. We will write a method in the \texttt{League} class called \texttt{showTeams()} to do this. The method will need to go through each team in the \texttt{teams} ArrayList and display the particular team’s information … perhaps using the \texttt{toString()} method from the \texttt{Team} class.

Hopefully, you “sense” that printing out all the teams involves repeating some code over and over again. That is, you should realize that we need a loop of some type. We have already discussed the \texttt{for} and \texttt{while} loops, but there is a special kind of \texttt{for} loop that is to be used when traversing through a collection such as an \texttt{ArrayList}. This loop is called the “\texttt{for-each}” loop, and its structure is a little simpler than the traditional \texttt{for} loop. Here is how we will use it to write the \texttt{showTeams()} method.

```java
void showTeams() {
    for (Team t: this.teams)
        System.out.println(t);  // or System.out.println(t.toString());
}
```

Notice that the \texttt{for-each} loop starts with \texttt{for} again, but this time the information within the round (\texttt{)} brackets is different. The format of this information is as follows. First we specify the \texttt{type} of object that is in the ArrayList … in this case \texttt{Team}. Then we specify a variable name which will be used to represent the particular team as we loop through them all … in this case we called it simply \texttt{t}. 
Then we use a colon : character followed by the name of the ArrayList that we want to loop through ... in this case this.teams. So, if we were to translate the for-each loop into English, it would sound something like this: “For each team t in the teams array list do the loop”.

Notice that within the loop, we simply use \textit{t} as we would use any other variable. In our example, \textit{t} is the \texttt{Team} object that we are examining during that round through the loop. So \textit{t} points to the 1\textsuperscript{st} team in the league when we begin the loop, then it points to the 2\textsuperscript{nd} team the next time through the loop, then the 3\textsuperscript{rd} team etc..

Let us test our method out using the following test program:

```java
class LeagueTestProgram {
   public static void main(String args[]) {
      League nhl;

      nhl = new League("NHL");

      //Add a pile of teams to the league
      nhl.teams.add(new Team("Ottawa Senators"));
      nhl.teams.add(new Team("Montreal Canadians"));
      nhl.teams.add(new Team("Toronto Maple Leafs"));
      nhl.teams.add(new Team("Vancouver Cannucks"));
      nhl.teams.add(new Team("Edmonton Oilers"));
      nhl.teams.add(new Team("Washington Capitals"));
      nhl.teams.add(new Team("New Jersey Devils"));
      nhl.teams.add(new Team("Detroit Red Wings"));

      //Display the teams
      System.out.println("\nHere are the teams:");
      nhl.showTeams();
   }
}
```

Here is the output so far:

```
Here are the teams:
The Ottawa Senators have 0 wins, 0 losses and 0 ties.
The Montreal Canadians have 0 wins, 0 losses and 0 ties.
The Toronto Maple Leafs have 0 wins, 0 losses and 0 ties.
The Vancouver Cannucks have 0 wins, 0 losses and 0 ties.
The Edmonton Oilers have 0 wins, 0 losses and 0 ties.
The Washington Capitals have 0 wins, 0 losses and 0 ties.
The New Jersey Devils have 0 wins, 0 losses and 0 ties.
The Detroit Red Wings have 0 wins, 0 losses and 0 ties.
```

Notice that all the teams have no recorded wins, losses or ties. Lets write a method that will record a win and a loss for two teams that play together, and another method to record a tie when the two teams play and tie.
void recordWinAndLoss(Team winner, Team loser) {
    winner.wins++;
    loser.losses++;
}

void recordTie(Team teamA, Team teamB) {
    teamA.ties++;
    teamB.ties++;
}

If we wanted to test these methods now, we could write test code like this:

League nhl;
Team team1, team2, team3;

nhl = new League("NHL");
nhl.teams.add(team1 = new Team("Ottawa Senators"));
nhl.teams.add(team2 = new Team("Montreal Canadians"));
nhl.teams.add(team3 = new Team("Toronto Maple Leafs"));

nhl.recordWinAndLoss(team1, team2);
nhl.recordTie(team1, team2);
nhl.recordWinAndLoss(team3, team2);
// ... etc ...

You should now notice something tedious. We would have to make variables for each team if we want to record wins, losses and ties among them. Why? Because the recording methods require Team objects ... the same Team objects that we added to the League ... so we would have to remember them ... hence requiring us to store them in a variable. Perhaps a better way to record wins, losses and ties would be to do something like this:

League nhl;

nhl = new League("NHL");
nhl.teams.add(new Team("Ottawa Senators"));
nhl.teams.add(new Team("Montreal Canadians"));
nhl.teams.add(new Team("Toronto Maple Leafs"));

nhl.recordWinAndLoss("Ottawa Senators", "Montreal Canadians");
nhl.recordTie("Ottawa Senators", "Montreal Canadians");
nhl.recordWinAndLoss("Toronto Maple Leafs", "Montreal Canadians");
// ... etc ...

This way, we do not need to create extra variables. However, we would have to make new recording methods that took Strings (i.e., the Team names) as parameters instead of Team
objects. Here are the methods that we would need to implement (notice the difference in the parameter types):

```java
void recordWinAndLoss(String winnerName, String loserName) {
    ...
}
void recordTie(String teamAName, String teamBName) {
    ...
}
```

To make this work, however, we still need to get into the appropriate Team objects and update their wins/losses/ties. Therefore, we will have to take the incoming team names and find the Team objects that correspond with those names. We would need to do this 4 times: once for the winnerName, once for the loserName, once for teamAName and once for teamBName. Rather than repeat the code 4 times, we will make a method to do this particular sub-task of finding a team with a given name. Here is the method that we will write:

```java
Team teamWithName(String nameToLookFor) {
    Team answer;
    ...
    return answer;
}
```

Notice that it will take the team’s name as a parameter and then return a Team object. How would we complete this method? We can use the for-each loop to traverse through all the teams and find the one with that name as follows:

```java
Team teamWithName(String nameToLookFor) {
    Team answer = null;
    for (Team t: this.teams) {
        if (t.name.equals(nameToLookFor))
            answer = t;
    }
    return answer;
}
```

Notice a few points. First, we set the answer to null. If we do not find a Team with the given name, the method returns null ... which is the only appropriate answer. Next, notice that for each team t, we compare its name with the incoming string aName and if these two strings are equal, then we have found the Team object that we want, so we store it in the answer variable to be returned at the completion of the loop.
This method can be shortened as follows:

```java
Team teamWithName(String nameToLookFor) {
    for (Team t: this.teams)
        if (t.name.equals(nameToLookFor))
            return t;
    return null;
}
```

Now that this method has been created, we can use it in our methods for recording wins/losses and ties as follows:

```java
void recordWinAndLoss(String winnerName, String loserName) {
    Team winner, loser;
    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    winner.wins++;
    loser.losses++;
}

void recordTie(String teamAName, String teamBName) {
    Team teamA, teamB;
    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    teamA.ties++;
    teamB.ties++;
}
```

The methods work as before, but there are potential problems. What if we cannot find the Team object with the given names (e.g., someone spelt the name wrong)? In this case, perhaps winner, loser, teamA or teamB will be null and we will get a NullPointerException when we try to access the team’s attributes. We can check for this with an if statement.

```java
void recordWinAndLoss(String winnerName, String loserName) {
    Team winner, loser;
    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    if ((winner != null) && (loser != null)) {
        winner.wins++;
        loser.losses++;
    }
}
```
void recordTie(String teamAName, String teamBName) {
    Team teamA, teamB;

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    if ((teamA != null) && (teamB != null)) {
        teamA.ties++;
        teamB.ties++;
    }
}

Now the games are only recorded when we have successfully identified the two Team objects that need to be updated as a result of the played game. Interestingly though, the same problem may occur in our previous recording methods … that is … the Team objects passed in may be null. Also, in our code, we already have method for recording the wins/losses/ties in the case where we have the Team objects, so we should call those methods from here. We can simply call the previous recording methods from these two new ones and move the null-checking in there instead as follows:

Team teamWithName(String nameToLookFor) {
    for (Team t: this.teams)
        if (t.name.equals(nameToLookFor))
            return t;
    return null;
}

void recordWinAndLoss(Team winner, Team loser) {
    if ((winner != null) && (loser != null)) {
        winner.wins++;
        loser.losses++;
    }
}

void recordTie(Team teamA, Team teamB) {
    if ((teamA != null) && (teamB != null)) {
        teamA.ties++;
        teamB.ties++;
    }
}

void recordWinAndLoss(String winnerName, String loserName) {
    Team winner, loser;

    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    this.recordWinAndLoss(winner, loser);
}

void recordTie(String teamAName, String teamBName) {
    Team teamA, teamB;

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    this.recordTie(teamA, teamB);
}
In fact, we can even shorten the last two methods by noticing that the variables are not really necessary:

```java
void recordWinAndLoss(String winnerName, String loserName) {
    this.recordWinAndLoss(this.teamWithName(winnerName),
                         this.teamWithName(loserName));
}

void recordTie(String teamAName, String teamBName) {
    this.recordTie(this.teamWithName(teamAName),
                    this.teamWithName(teamBName));
}
```

Consider a method called `totalGamesPlayed()` which is supposed to return the total number of games played in the league. All we need to do is count the number of games played by all the teams (i.e., we will need some kind of counter) and then divide by 2 (since each game was played by two teams, hence counted twice). Here is the format:

```java
int totalGamesPlayed() {
    int total = 0;
    ...
    return total/2;
}
```

We will also need a `for-each` loop to go through each team:

```java
int totalGamesPlayed() {
    int total = 0;
    for (Team t: this.teams) {
        ...
        return total/2;
    }
}
```

Now, if you were to look back at the `Team` class, you would notice a method in there called `gamesPlayed()`. That means, we can ask a team how many games they played by simply calling that method. We should be able to make use of this value as follows:

```java
int totalGamesPlayed() {
    int total = 0;
    for (Team t: this.teams)
        total += t.gamesPlayed();

    return total/2;
}
```
Notice that the method is quite simple, as long as you break it down into simple steps like we just did. For more practice, let us find the team that is in first place (i.e., the Team object that has the most points). We can start again as follows:

```java
Team firstPlaceTeam() {
    Team teamWithMostPoints = null;
    ...
    return teamWithMostPoints;
}
```

Notice that it returns a Team object. Likely, you realize that we also need a for-each loop since we need to check all of the teams:

```java
Team firstPlaceTeam() {
    Team teamWithMostPoints = null;
    for (Team t: this.teams) {
        ...
    }
    return teamWithMostPoints;
}
```

Again, we can make use of a pre-defined method in the Team class called totalPoints() which returns the number of points for a particular team:

```java
Team firstPlaceTeam() {
    int points;
    Team teamWithMostPoints = null;
    for (Team t: this.teams) {
        points = t.totalPoints();
    }
    return teamWithMostPoints;
}
```

But now what do we do? The current code will simply grab each team’s point values one at a time. We need to somehow compare them. Many students have trouble breaking this problem down into simple steps. The natural tendency is to say to yourself “I will compare the 1st team’s points with the 2nd team’s points and see which is greater”. If we do this however, then what do we do with that answer? How does the third team come into the picture?

Hopefully, after some thinking, you would realize that as we traverse through the teams, we need to keep track of (i.e., remember) the best one so far.
Imagine for example, searching through a basket of apples to find the best one. Would you not grab an apple and hold it in your hand and then look through the other apples and compare them with the one you are holding in your hand? If you found a better one, you would simply trade the one currently in your hand with the new better one. By the time you reach the end of the basket, you are holding the best apple.

Well we are going to do the same thing. The `teamWithMostPoints` variable will be like our good apple that we are holding. Whenever we find a team that is better (i.e., more points) than this one, then that one becomes the `teamWithMostPoints`. Here is the code:

```java
Team firstPlaceTeam() {
    Team teamWithMostPoints = null;
    for (Team t: this.teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}
```

Does it make sense? There is one small issue though. Just like we need to begin our apple checking by picking up a first apple, we also need to pick a team (any `Team` object) to be the “best” one before we start the search. Currently the `teamWithMostPoints` starts off at `null` so we need to set this to a valid `Team` so start off. We can perhaps take the first `Team` in the `teams` ArrayList:

```java
Team firstPlaceTeam() {
    Team teamWithMostPoints = this.teams.get(0);
    for (Team t: this.teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}
```

We are not done yet! It is possible, in a weird scenario, that there are no teams in the league! In this case `teams.get(0)` will return `null` and we will get a `NullPointerException` again when we go to ask for the `totalPoints()`. So, we would need to add a special case to return `null` if the `teams` list is empty. Here is the new code …
Team firstPlaceTeam() {
    Team teamWithMostPoints;

    if (this.teams.size() == 0)
        return null;

    teamWithMostPoints = this.teams.get(0);
    for (Team t: this.teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }
    return teamWithMostPoints;
}

What would we change in the above code if we wanted to write a method called 
lastPlaceTeam() that returned the team with the least number of points? Try to do it.

For the purpose of a summary, here is the entire League class as we have defined it:

import java.util.ArrayList;

class League {
    String name;
    ArrayList<Team> teams;

    League(String n) {
        this.name = n;
        this.teams = new ArrayList<Team>(); // Doesn’t make any Team objects
    }

    // This specifies the appearance of the League
    public String toString() { return "The " + this.name + " league"; } }

    // Display all the teams to the console
    void showTeams() {
        for (Team t: this.teams)
            System.out.println(t); // or System.out.println(t.toString());
    }

    // Find and return the Team object that has the given name, null if not found
    Team teamWithName(String nameToLookFor) {
        for (Team t: this.teams)
            if (t.name.equals(nameToLookFor))
                return t;
        return null;
    }

    // Record a win for team winner and loss for team loser
    void recordWinAndLoss(Team winner, Team loser) {
        if ((winner != null) && (loser != null)) {
            winner.wins++;
            loser.losses++;
        }
    }
}
// Record a tie for each of TeamA and TeamB
void recordTie(Team teamA, Team teamB) {
    if ((teamA != null) && (teamB != null)) {
        teamA.ties++;
        teamB.ties++;
    }
}

// Record a win for the team with name winnerName
// and a loss for the team with name loserName
void recordWinAndLoss(String winnerName, String loserName) {
    this.recordWinAndLoss(this.teamWithName(winnerName),
                          this.teamWithName(loserName));
}

// Record a tie for the two teams with the given names
void recordTie(String teamAName, String teamBName) {
    this.recordTie(this.teamWithName(teamAName),
                    this.teamWithName(teamBName));
}

// Return the total number of games played in the league
int totalGamesPlayed() {
    int total = 0;
    for (Team t: this.teams)
        total += t.gamesPlayed();

    return total/2;
}

// Return the team that has the most points
Team firstPlaceTeam() {
    Team teamWithMostPoints;

    if (this.teams.size() == 0)
        return null;

    teamWithMostPoints = this.teams.get(0);
    for (Team t: this.teams) {
        if (t.totalPoints() > teamWithMostPoints.totalPoints())
            teamWithMostPoints = t;
    }

    return teamWithMostPoints;
}

// Return the team that has the least points
Team lastPlaceTeam() {
    Team teamWithLeastPoints;

    if (this.teams.size() == 0)
        return null;

    teamWithLeastPoints = this.teams.get(0);
    for (Team t: this.teams) {
        if (t.totalPoints() < teamWithLeastPoints.totalPoints())
            teamWithLeastPoints = t;
    }

    return teamWithLeastPoints;
}
Here is a program that can be used to test our methods:

```java
class LeagueTestProgram2 {
    public static void main(String args[]) {
        League nhl = new League("NHL");

        // Add a pile of teams to the league
        nhl.teams.add(new Team("Ottawa Senators"));
        nhl.teams.add(new Team("Montreal Canadians"));
        nhl.teams.add(new Team("Toronto Maple Leafs"));
        nhl.teams.add(new Team("Vancouver Cannucks"));
        nhl.teams.add(new Team("Edmonton Oilers"));
        nhl.teams.add(new Team("Washington Capitals"));
        nhl.teams.add(new Team("New Jersey Devils"));
        nhl.teams.add(new Team("Detroit Red Wings"));

        // Now we will record some games
        nhl.recordWinAndLoss("Ottawa Senators", "New Jersey Devils");
        nhl.recordWinAndLoss("Edmonton Oilers", "Montreal Canadians");
        nhl.recordTie("Ottawa Senators", "Detroit Red Wings");
        nhl.recordWinAndLoss("Montreal Canadians", "Washington Capitals");
        nhl.recordWinAndLoss("Ottawa Senators", "Edmonton Oilers");
        nhl.recordTie("Washington Capitals", "Edmonton Oilers");
        nhl.recordWinAndLoss("Vancouver Cannucks", "Toronto Maple Leafs");
        nhl.recordWinAndLoss("Toronto Maple Leafs", "Edmonton Oilers");
        nhl.recordWinAndLoss("New Jersey Devils", "Detroit Red Wings");

        // This one will not work
        nhl.recordWinAndLoss("Mark's Team", "Detroit Red Wings");

        // Now display the teams again
        System.out.println("Here are the teams after recording the " +
            "wins, losses and ties:
        nhl.showTeams();

        // Here are some statistics
        System.out.println("The total number of games played is " +
            nhl.totalGamesPlayed());
        System.out.println("The first place team is " +
            nhl.firstPlaceTeam());
        System.out.println("The last place team is " +
            nhl.lastPlaceTeam());
    }
}
```

Here would be the output (make sure that it makes sense to you) …
Here are the teams after recording the wins, losses and ties:

The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The Montreal Canadiens have 1 wins, 1 losses and 0 ties.
The Toronto Maple Leafs have 1 wins, 1 losses and 0 ties.
The Vancouver Cannucks have 1 wins, 0 losses and 0 ties.
The Edmonton Oilers have 1 wins, 2 losses and 1 ties.
The Washington Capitals have 0 wins, 1 losses and 1 ties.
The New Jersey Devils have 1 wins, 1 losses and 1 ties.
The Detroit Red Wings have 0 wins, 1 losses and 2 ties.

The total number of games played is 10
The first place team is The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The last place team is The Washington Capitals have 0 wins, 1 losses and 1 ties.

Supplemental Information

There is an additional class called Vector which has the same functionality as the ArrayList class. In fact, in most situations, you can simply replace the word ArrayList by Vector and your code will still compile. There is a small difference between ArrayLists and Vectors. They have the same functionality, but ArrayLists are faster because they have methods that are not synchronized. Vectors allow multiple processes (or multiple "programs") to access/modify them at the same time, so they have extra code in the methods to ensure that the Vector is shared properly and safely between the processes. We will not talk any more about this in this course. You should always use ArrayLists when creating simple programs.

6.4 Car/Autoshow Example

Many agree that the best way of learning something is by example. Therefore, let us consider another example that uses ArrayLists. We will create an Autoshow that contains many Car objects in an ArrayList. Although there are many pieces of information that we may want to keep track of for a Car, lets consider just keeping track of the car’s make, model, color, topSpeed and whether or not it has4Doors (we will assume that all cars are either 2-door or 4-door, ignoring 3-door and 5-door cars).

Here is a simple class definition showing the attributes (i.e., instance variables), a constructor and a toString() method ...
class Car {
    String make;    // name of company that makes it (e.g., Honda)
    String model;   // name of car itself (e.g., Civic)
    String color;   // color (e.g., red)
    int topSpeed;   // fastest that the car goes
    boolean has4Doors;   // true if it has 4 doors, false otherwise

    // Here is a constructor
    Car(String mk, String md, String cl, int ts, boolean fd) {
        this.make = mk;
        this.model = md;
        this.color = cl;
        this.topSpeed = ts;
        this.has4Doors = fd;
    }

    // This method returns a string representing the car in this format:
    // "Red 2-door Porsche 911 with top speed 340kmph"
    public String toString() {
        String s = this.color;
        if (this.has4Doors)
            s += " 4-door ";
        else
            s += " 2-door ";
        return (s + this.make + " " + this.model +
            " with top speed " + this.topSpeed + "kmph");
    }
}

Now what about the Autoshow? It will likely have a lot of information (e.g., name, start date, end date, admission prices, etc.). However, we will keep things simple and just make use of two attributes... a name and a list of cars:

import java.util.ArrayList;

class Autoshow {
    // These are the instance variables
    String name;  // Name of autoshow (e.g., "AutoRama 2009")
    ArrayList<Car> cars;  // The cars at the show

    // Here is a constructor
    Autoshow(String n) {
        this.name = n;
        this.cars = new ArrayList<Car>();
    }

    // This method returns a string representing the autoshow
    public String toString() {
        return ("Autoshow " + this.name + " with " +
            this.cars.size() + " cars");
    }
}
Notice that the **cars** list is specified to hold `<Car>` objects.

Suppose that we wanted to create a method that prints all the “cars with a given make” (e.g., print out all **Honda** cars). How can we do this? Well, you should always begin by specifying the name of the method, its parameters and its return type. We will call the method `carsWithMake()` and it should take-in (as a parameter) a **String** which specifies the make of the cars we are looking for.

```java
void carsWithMake(String m) {
    ...
}
```

As you may have guessed, in order to make a list of all the companies that have cars at the autoshow, we will need to check out all of the cars. So, we will need to loop through them. Here is a strategy that prints out all the cars:

```java
for (Car c: this.cars) {
    System.out.println(c);
}
```

To print out ONLY those cars that have make `m`, we need to use an **if** statement. Remember, we use the `.equals()` method to compare the makes since they are **String** objects, not primitives:

```java
void carsWithMake(String m) {
    for (Car c: this.cars) {
        if (c.make.equals(m))
            System.out.println(c);
    }
}
```

Now instead of **printing** all the cars with a given make, how would we adjust the method to **return a list** of all the cars with a specific make? We would need to return a list of **Car** objects … hence we need to change the return type from **void** to **ArrayList<Car>** as follows:

```java
ArrayList<Car> carsWithMake(String m) {
    ...
}
```

Then, instead of printing the cars out, we would need to create a new list (to hold the answer), fill that list up with all the cars that have the specified make, and then return that list …
ArrayList<Car> carsWithMake(String m) {
    ArrayList<Car> answer;
    answer = new ArrayList<Car>(); // make the empty answer list
    for (Car c: this.cars) {
        if (c.make.equals(m))
            answer.add(c); // add the car to the answer list
    }
    return answer;
}

Now let us make another method that returns all of the "different makes of cars" at the auto show. Of course, we want only a list of the different makes, so no duplicates are allowed. Again we need to return an ArrayList, but this time it will contain String objects, since we want the makes of the cars, not the cars themselves:

ArrayList<String> differentMakes() {
    ArrayList<String> answer = new ArrayList<String>();
    ... 
    return answer;
}

Again, we need to loop through all the cars. This time though, we need to get the make of the car and add it to the list:

ArrayList<String> differentMakes() {
    ArrayList<String> answer = new ArrayList<String>();
    for (Car c: this.cars) {
        if (c.make.equals(m))
            answer.add(c); // add the car to the answer list
    }
    return answer;
}

This code works. However, it will show duplicates. That is, if there are 10 cars made by "Honda", then "Honda" will appear 10 times in the answer. Likely, we only want a list of unique makes. To do this, we need to determine whether or not the make has already been added to the answer. If it has, then we don’t add it again.

The JAVA “guys” knew that there would often be a need to ask whether or not an ArrayList contains a particular item already. So, they created a method called contains() which allows us to ask whether or not a particular Object is already in the list. The contains(Object x) method returns a boolean value of true if the object x is in the ArrayList and false otherwise. Here is how we can use it for our purposes …
ArrayList<String> differentMakes() {
    ArrayList<String> answer = new ArrayList<String>();
    for (Car c: this.cars) {
        if (!answer.contains(c.make))
            answer.add(c.make);
    }
    return answer;
}

Notice that we ask the answer ArrayList if it contains Car c’s make. If it does not (i.e., notice the ! character before the word answer), then the make is added to the list. This completes our task.

How about writing a method to find the "fastest car"? This would be the car with the maximum topSpeed. Well, as before, we realize that we need to go through all the cars, so we start out with the method signature and looping structure. The only difference is that the return type is a single Car now, not an ArrayList. Of course, if there was a "tie" for "fastest car" (i.e., two or more with the same topSpeed), then this method would only return one of the cars:

Car fastestCar() {
    Car fastest = null;
    for (Car c: this.cars) {
        // ... don't know what goes here yet ...
    }
    return fastest;
}

To complete the method, we follow the same “template” that we used to determine the firstPlaceTeam() in our League class. Do you remember? We keep track of the fastest car so far and replace it whenever we find a faster one. Here is the code:

Car fastestCar() {
    Car fastest = null;
    for (Car c: this.cars) {
        if (c.topSpeed > fastest.topSpeed)
            fastest = c;
    }
    return fastest;
}

Remember though, that we must start off with a valid Car object, not with null …
Car fastestCar() {
    if (this.cars.isEmpty())
        return null;

    Car fastest = this.cars.get(0);

    for (Car c: this.cars) {
        if (c.topSpeed > fastest.topSpeed)
            fastest = c;
    }

    return fastest;
}

Notice that there is an isEmpty() method for ArrayLists that returns true if the list is empty and false otherwise. It has the same effect as asking whether size() == 0.

Now let us see if we can print out the cars in order of their top speed … fastest car shown first. Assume that cars with the same speed will appear in arbitrary order among themselves. Here is what we will start with:

```java
void printBySpeed() {
    // ... don't know what goes here yet ...
}
```

You may feel that perhaps we can repeatedly find the fastest car... using our fastestCar() method. But why won't the following code work:

```java
void printBySpeed() {
    for (Car c: this.cars) {
        System.out.println(this.fastestCar());
    }
}
```

If you guessed that the code above would always find the same car over and over again … then you would be correct. What we need to do is find the fastest car, then remove it, then find the fastest car again from the remaining ones ... doing this until there are no more cars.

ArrayLists happen to have a method called remove(x) that allows you to remove an object x from the list, (provided that x is in the list to begin with). We can remove a car c from our cars ArrayList by doing this: this.cars.remove(c). We can replace c by a call to this.fastestCar() and obtain the following code …
void printBySpeed() {
    Car f;
    for (Car c: this.cars) {
        f = this.fastestCar();
        System.out.println(f);
        this.cars.remove(f);
    }
}

Intuitively, this would work fine. However, this code will not work. The reason is that JAVA prevents us from removing from an ArrayList while we are looping through it. JAVA will generate a ConcurrentModificationException. ... which means that we are trying to modify (i.e., remove from) the ArrayList while we are at the same time (i.e., concurrently) indexing through it. The problem is that the for loop makes use of the indices of the items in the list while it iterates through them ... and when we remove an item in the middle of the list all the items after it are shifted backwards, thereby changing their individual indices. This messes up the iterative for loop process.

We can fix this however, by using a while loop since the while loop does not keep track of indices. That is, we can repeatedly find and remove the fastest car while the list is not empty. Here is the code:

void printBySpeed() {
    Car f;
    while (!this.cars.isEmpty()) {
        f = this.fastestCar();
        System.out.println(f);
        this.cars.remove(f);
    }
}

This code will work as expected ... but it does something undesirable. It actually removes ALL the cars from the autoshow! That means, after this method is called, our autoshow has been ruined. One idea to “fix” this is to make a copy of the cars list and then loop through and remove from the copy, thereby leaving the original list intact.

We can make a copy of an ArrayList by calling one of its constructors that allows us to pass in an existing ArrayList as follows:

ArrayList<Car> copy;
copy = new ArrayList<Car>(this.cars);

This code will create a copy of the cars list that will contain the same cars as the original list ...
Now if we remove cars from the copy, we are merely removing the pointers (or arrows) to the cars, so the original cars list remains intact. Here is how we can adjust our code:

```java
void printBySpeed() {
    Car f;
    ArrayList<Car> copy;
    copy = new ArrayList<Car>(this.cars);
    while (!copy.isEmpty()) {
        f = this.fastestCar();
        System.out.println(f);
        copy.remove(f);
    }
}
```

However, this does not work either! Why not? Well, we are finding the fastest car from the original list, but removing from the copy. We need to find the fastest car in the copy. However, our `fastestCar()` method looks at the `this.cars` list, not the `copy`. So we cannot simply call the `fastestCar()` method since it does not use the `copy`.

What we CAN do is extract the code from the `fastestCar()` method and paste it in here, changing references to `this.cars` into `copy` as follows …
```java
void printBySpeed() {
    Car f;
    ArrayList<Car> copy;

    copy = new ArrayList<Car>(this.cars);
    while (!copy.isEmpty()) {
        f = copy.get(0);
        for (Car c: copy) {
            if (c.topSpeed > f.topSpeed)
                f = c;
        }
        System.out.println(f);
        copy.remove(f);
    }
}
```

Now we have the solution that we needed. Of course, you should realize that there are many ways to approach this problem. I’m sure you may have your own ideas as to how to solve the problem.

In a similar way, we can make a minor adjustment to write a method that "prints out all the cars in alphabetical order of their make". Cars with the same make will appear in arbitrary order amongst themselves. To do this, we need to know how to compare two strings to determine if one comes alphabetically before another.

In JAVA’s String class, there is a method called `s1.compareTo(s2)` that compares string `s1` with string `s2`. If `s1` comes before `s2` alphabetically, the result of this method is a negative number. If `s1` comes after `s2` alphabetically, then the result is a positive number. Otherwise if `compareTo()` returns zero, this indicates that `s1` equals `s2`. Here is how we can use this method on our previous code to solve our new problem:

```java
void printByMake() {
    Car f;
    ArrayList<Car> copy;

    copy = new ArrayList<Car>(this.cars);
    while (!copy.isEmpty()) {
        f = copy.get(0);
        for (Car c: copy) {
            if (c.make.compareTo(f.make) < 0)
                f = c;
        }
        System.out.println(f);
        copy.remove(f);
    }
}
```
The solution is simple, provided that you knew that there was a `compareTo()` method available to you. It is a good idea that you start looking around in the JAVA documentation for useful methods whenever you are stuck on something or need some help in manipulating `Strings` or `ArrayLists` or any type of object. Just double click on a word (usually a class name) in your JCreator editor and then select JDK Help under the Help menu. This will bring up the JAVA documentation for the object that you clicked on … showing you various methods/tools that you can apply to that object.

For a bigger challenge how could we write a method called `mostCommonColor()` that returns the most common color of all the cars at the autoshow? At first, it seems difficult. But we can simply begin by getting the method signature correct, returning a `String` and inserting our `for` loop … because we know that we need to check all of the cars:

```java
String mostCommonColor() {
    String bestColorSoFar = "";
    for (Car c: this.cars) {
        // update the bestColorSoFar when we find a more common one ...
    }
    return bestColorSoFar;
}
```

Now how do we approach the problem? Well think of real life. If your friend asked you to go to the autoshow and come back and tell him/her what the most popular color was at the show, how would YOU go about doing this?

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…:
If we were to take this approach in our programming solution, then we need to realize that each color that we write down on our clipboard will have a single number associated with it at all times (representing the count of the number of times that color appeared).

The clipboard/paper itself represents our list of colors so we would need an ArrayList to store that. In fact, since we need a list of counts along with the list of colors, we will need two ArrayLists … one to store the colors and one to store the counts. So, here is what we have so far:

```java
String mostCommonColor() {
    String bestColorSoFar = "";
    ArrayList<String> colors = new ArrayList<String>();
    ArrayList<Integer> counts = new ArrayList<Integer>();
    for (Car c: this.cars) {
        // update the bestColorSoFar when we find a more common one...
    }
    return bestColorSoFar;
}
```

Now let's see how we can go through the cars and fill up the colors list. This is done exactly the same way as we did the differentMakes() method … but checking the color attribute instead of the make attribute:

```java
String mostCommonColor() {
    String bestColorSoFar = "";
    ArrayList<String> colors = new ArrayList<String>();
    ArrayList<Integer> counts = new ArrayList<Integer>();
    for (Car c: this.cars) {
        if (colors.contains(c.color)) {
            // don’t add it to the list
        }
        else {
            colors.add(c.color);  // append color to colors list
            counts.add(1);  // append count value to counts list
        }
    }
    return bestColorSoFar;
}
```

Now we need to adjust our code so that it also updates the counter associated with the color as we find it. In the case where it is a new color (i.e., the else code above), then this is easy … we simply add a 1 to the end of the counts ArrayList:

```java
else {
    colors.add(c.color);  // append color to colors list
    counts.add(1);  // append count value to counts list
}
```
The above code adds the color/count pair each time a new color is found. This works fine in our little example above when we find the 1st **red** car and then the 1st **white** car, but what happens when we find the 2nd **white** car? We need to go through our **colors** list and find the color **white**, then change its corresponding **counts** value from 1 to 2. To do this, we will need to make use of a couple of more **ArrayList** methods.

As mentioned earlier, ArrayLists keep their values ordered by index … which is an integer indicating the position of the value within the list. So, in our example, once we encounter the first **red** car and then the first **white** car, the two lists look as shown here →

Notice that the color **red** is at index 0 of the **colors** ArrayList and **white** is at index 1. In the **counts** list, the corresponding count values are 1 and 1. When encountering the 2nd **white** car, we need to determine the index of **white** in the colors list (which is 1) and then get the item at this index in the **counts** list and increase it by 1.

ArrayLists have a useful method to get the index of a value called **indexOf()**. We can ask the **colors** list for the **indexOf**("**White"”) and it will return the location of **white** in the list (in this case 1).

Then we can use the **get(1)** method in the **counts** list to get the current **white** car count (in this case 1) and add 1 to it to make it 2. However, in order to make a change to the **counts** list we have to make use of the ArrayList **set(x, y)** method which will allow us to set the value at position x in the list to be y. By means of a diagram, here is what we need to do →

So, we are making use of the **indexOf()**, **get()** and **set()** methods.
Here is what we now have:

```java
String mostCommonColor() {
    int loc;
    String bestColorSoFar = "";
    ArrayList<String> colors = new ArrayList<String>();
    ArrayList<Integer> counts = new ArrayList<Integer>();

    for (Car c: this.cars) {
        if (colors.contains(c.color)) {
            loc = colors.indexOf(c.color);
            counts.set(loc, counts.get(loc) + 1);
        } else {
            colors.add(c.color);
            counts.add(1);
        }
    }
    return bestColorSoFar;
}
```

At this point, our code keeps track of all the unique colors found at the autoshow as well as the corresponding count for that color. Now we just need to determine which is the most common color.

I am sure you will agree that the answer is the color whose corresponding count is highest (in the case of a tie, we will just return the first color encountered that has this highest value).

To do this, we take the approach that we used in our `fastestCar()` method … that is … keep track of the highest count as we move along the autoshow, updating this highest value whenever we encounter a higher count. Here is the code …

```java
String mostCommonColor() {
    int loc;
    int bestCountSoFar = 0;
    String bestColorSoFar = "";
    ArrayList<String> colors = new ArrayList<String>();
    ArrayList<Integer> counts = new ArrayList<Integer>();

    for (Car c: this.cars) {
        if (colors.contains(c.color)) {
            loc = colors.indexOf(c.color);
            counts.set(loc, counts.get(loc) + 1);
            if (counts.get(loc) > bestCountSoFar) {
                bestCountSoFar = counts.get(loc);
                bestColorSoFar = c.color;
            }
        } else {
            colors.add(c.color);
            counts.add(1);
        }
    }
    return bestColorSoFar;
}
```
else {
    colors.add(c.color);
    counts.add(1);
    if (1 > bestCountSoFar) {
        bestCountSoFar = 1;
        bestColorSoFar = c.color;
    }
}
}
return bestColorSoFar;

We can simplify the code a little by noticing that the recently added code is similar for both cases. If we add a newCount variable to hold the new updated count value for the item just added to the list, then we can simplify the code as follows:

```java
String mostCommonColor() {
    int loc, newCount;
    int bestCountSoFar = 0;
    String bestColorSoFar = "";
    ArrayList<String> colors = new ArrayList<String>();
    ArrayList<Integer> counts = new ArrayList<Integer>();
    for (Car c: this.cars) {
        if (colors.contains(c.color)) {
            loc = colors.indexOf(c.color);
            newCount = counts.get(loc) + 1;
            counts.set(loc, newCount);
        } else {
            colors.add(c.color);
            newCount = 1;
            counts.add(newCount);
        }
        if (newCount > bestCountSoFar) {
            bestCountSoFar = newCount;
            bestColorSoFar = c.color;
        }
    }
    return bestColorSoFar;
}
```

Well, that is it. The code should work as planned … returning the most common color at the autoshow.
We have written quite a few methods. Here is a test program to try them all out...

class AutoshowTestProgram {
    public static void main(String args[]) {
        Autoshow show = new Autoshow("AutoRama 2009");

        // First add lots of cars to the show
        show.cars.add(new Car("Porsche", "959", "Red", 240, false));
        show.cars.add(new Car("Ford", "Mustang", "White", 230, false));
        show.cars.add(new Car("Volkswagen", "Beetle", "Blue", 140, false));
        show.cars.add(new Car("Volkswagen", "Jetta", "Silver", 180, true));
        show.cars.add(new Car("Geo", "Storm", "Yellow", 110, true));
        show.cars.add(new Car("Ford", "Escort", "Yellow", 10, true));
        show.cars.add(new Car("Honda", "Civic", "Black", 220, true));
        show.cars.add(new Car("Nissan", "Altima", "Silver", 180, true));
        show.cars.add(new Car("BMW", "5", "Gold", 260, true));
        show.cars.add(new Car("Prelude", "Honda", "White", 90, false));
        show.cars.add(new Car("Mazda", "RX7", "Red", 240, false));
        show.cars.add(new Car("Mazda", "MX6", "Green", 160, true));
        show.cars.add(new Car("Pontiac", "G6", "Black", 140, false));

        // Now test our fun methods
        System.out.println(show);
        System.out.println("Here are the Pontiac cars:");
        System.out.println(show.carsWithMake("Pontiac");
        System.out.println("Here are the Ford cars:");
        System.out.println(show.carsWithMake("Ford");
        System.out.println("Here are the different makes:");
        System.out.println(show.differentMakes());
        System.out.println("This is the fastest car:");
        System.out.println(show.fastestCar());
        System.out.println("The most common color is " + show.mostCommonColor());
        System.out.println("\nHere are the cars sorted by top speed:");
        show.printBySpeed();
        System.out.println("\nHere are the cars sorted by make:");
        show.printByMake();
    }
}
Here are the test results:

Autoshow AutoRama 2009 with 15 cars
Here are the Pontiac cars:
[White 4-door Pontiac Grand-Am with top speed 160kmph, Black 2-door Pontiac G6 with top speed 140kmph]
Here are the Ford cars:
[White 2-door Ford Mustang with top speed 230kmph, Yellow 4-door Ford Escort with top speed 10kmph]
Here are the different makes:
[Porsche, Pontiac, Ford, Volkswagon, Geo, Toyota, Honda, Nissan, BMW, Prelude, Mazda]
This is the fastest car:
Gold 4-door BMW 5 with top speed 260kmph
The most common color is White

Here are the cars sorted by top speed:
Gold 4-door BMW 5 with top speed 260kmph
Red 2-door Porsche 959 with top speed 240kmph
Red 2-door Mazda RX7 with top speed 240kmph
White 2-door Ford Mustang with top speed 230kmph
Black 2-door Toyota MR2 with top speed 220kmph
Black 4-door Honda Civic with top speed 220kmph
Silver 4-door Volkswagon Jetta with top speed 180kmph
Silver 4-door Nissan Altima with top speed 180kmph
White 4-door Pontiac Grand-Am with top speed 160kmph
Green 4-door Mazda MX6 with top speed 160kmph
Blue 2-door Volkswagon Beetle with top speed 140kmph
Black 2-door Pontiac G6 with top speed 140kmph
Yellow 4-door Geo Storm with top speed 110kmph
White 2-door Prelude Honda with top speed 90kmph
Yellow 4-door Ford Escort with top speed 10kmph

Here are the cars sorted by make:
Gold 4-door BMW 5 with top speed 260kmph
White 2-door Ford Mustang with top speed 230kmph
Yellow 4-door Ford Escort with top speed 10kmph
Yellow 4-door Geo Storm with top speed 110kmph
Black 4-door Honda Civic with top speed 220kmph
Red 2-door Mazda RX7 with top speed 240kmph
Green 4-door Mazda MX6 with top speed 160kmph
Silver 4-door Nissan Altima with top speed 180kmph
White 4-door Pontiac Grand-Am with top speed 160kmph
Black 2-door Pontiac G6 with top speed 140kmph
Red 2-door Porsche 959 with top speed 240kmph
White 2-door Prelude Honda with top speed 90kmph
Black 2-door Toyota MR2 with top speed 220kmph
Blue 2-door Volkswagon Beetle with top speed 140kmph
Silver 4-door Volkswagon Jetta with top speed 180kmph