Chapter 8

Abstract Data Types

What is in This Chapter ?

In this chapter we discuss the notion of Abstract Data Types (ADTs) as they pertain to storing collections of data in our programs. There are many common ADTs used in computer science. We will discuss here some of the common ones such as **Lists**, **Queues**, **Deques**, **Linked-Lists**, **Stacks**, **Sets** and **Dictionaries**. You will understand the differences between these various ADTs in terms of the operations that you can perform on them. Lastly, we will implement a Doubly-Linked Lists data structure to help you understand how pointers can be used to define a recursive data structure.

8.1 Common Abstract Data Types

Every time we define a new object, we are actually defining a new data type. That is, we are grouping attributes and behaviors to form a new type of data (i.e., object) we can use throughout our programs as if it were a single piece of data. There are actually some commonly used models for defining similar types of data:

> *An **abstract data type (ADT)** is a mathematical model for a certain class of data structures that have similar behavior. (Wikipedia)*

The word *abstract* here means that we are discussing data types in a general manner, without having a particular practical purpose or intention in mind. There are different types of ADTs, each with their own unique way for storing, accessing and modifying the data. Typically, ADTs will store general data of any kind, although usually the data inside the ADT is all of the same kind ... or at least has something in common.

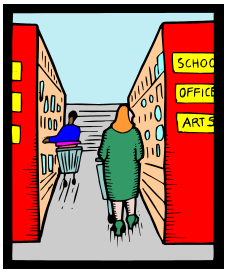
ADTs are a vital part of any programming language since they are used to *collect* data together in an "easy-to-use" way. We often use the term ***collection*** to represent one of these data types. There are advantages of using ADTs:

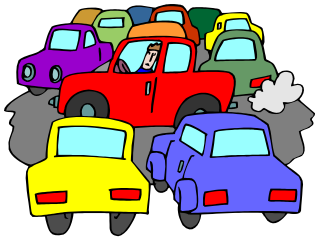
1. They help to simplify descriptions of abstract algorithms, thereby allowing us to write simplified pseudocode with less details (e.g., we can write pseudocode such as "**Add x to the list**" instead of "**Put x at position size in the array and then let size = size+1**").

2. They allow us to classify and evaluate data structures in regards to the common behaviors between data types (e.g., one ADT may have a more efficient *remove* operation while another may have a more efficient *add* or *search* operation. We could choose the ADT that best fits our needs).

As we have already seen with **Arrays**, collections allow many objects to be collected/stored together and then passed around as a single object (i.e., the array itself is an object). Just about every useful application of any kind requires collections for situations such as:

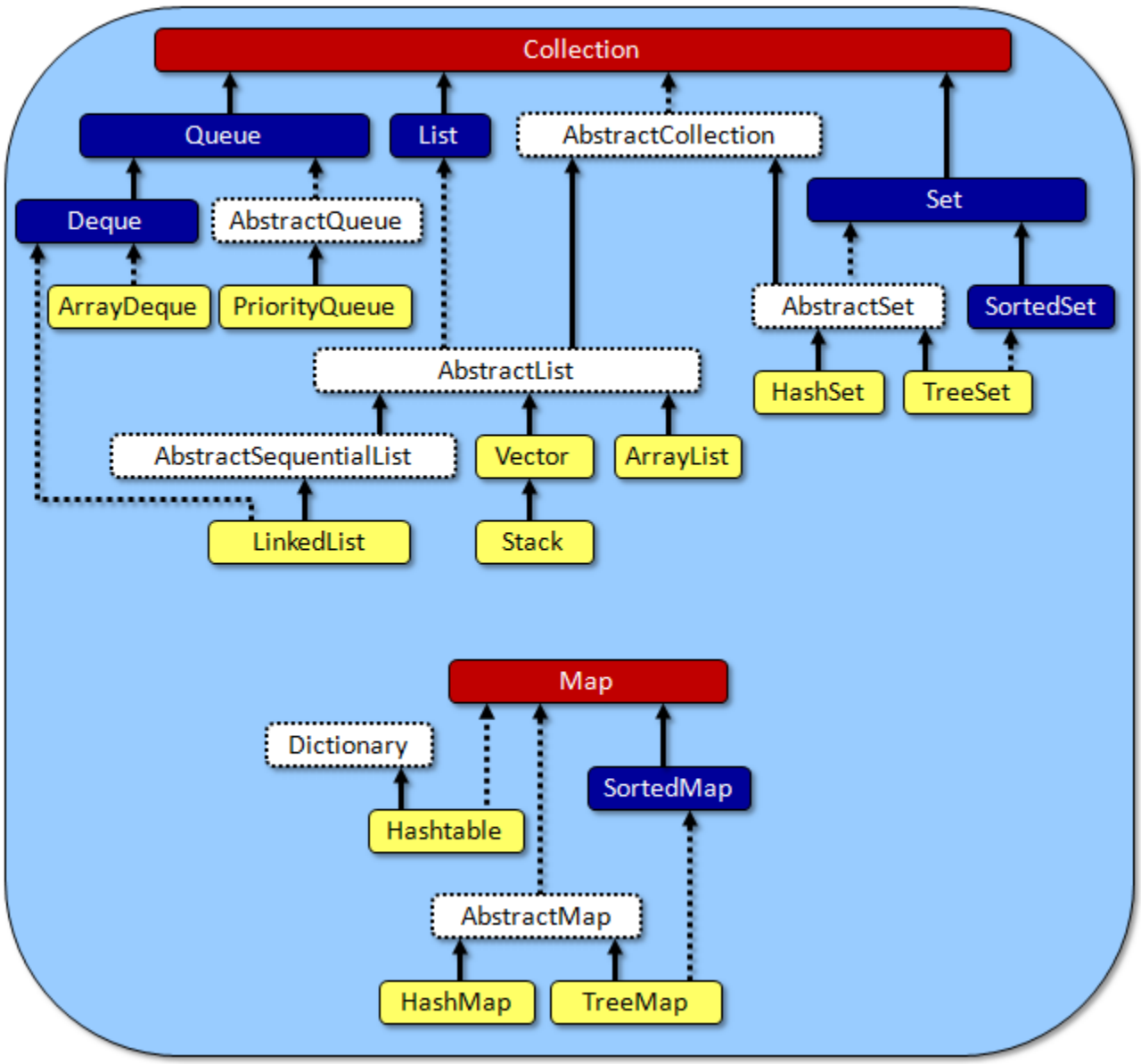
- storing products on shelves in a store
- maintaining information on customers
- keeping track of cars for sale, for rental or for servicing
- a personal collection of books, CDs, DVDs, cards, etc...
- maintaining a shopping cart of items to be purchased from a website





In JAVA, there are a variety of ADT-related classes that can be used to represent these various programming needs. These ADTs are located in the **java.util.Collection** package, along with some other useful tools. In this set of notes, we investigate (very briefly) some of these JAVA collections in a way that will help a programmer understand which ADT is best for their particular programming application.

ADTs in JAVA are organized into a "seemingly complicated" hierarchy of JAVA interfaces and classes. There are two sub-hierarchies ... one is rooted at **Collection**, the other is rooted at **Map**. Here is a diagram showing part of this hierarchy:



In the above hierarchy, the red and dark blue represent java **interfaces**, the white represents **abstract** classes and the yellow represents **concrete** classes. The solid arrows indicate **inheritance** while the dashes lines indicate that a class **implements** an interface. We will be discussing some of these classes in detail. You may want to refer back to this diagram once in a while to ensure that you understand how the classes differ and how they are similar.

Notice that there are 11 concrete classes, and that all of them indirectly implement the **Collection** or **Map** interface. Recall that an *interface* just specifies a list of method signatures ... not any code. That means, all of the concrete collection & map classes have something in common and that they all implement a common set of methods.

The main commonality between the collection classes is that they all store objects called their ***elements***, which may be heterogeneous objects (i.e., the elements may be a mix of various (possibly unrelated) objects). Storing mixed kinds of objects in a **Collection** is allowed, but not often done unless there is something in common with the objects (i.e., they extend a common **abstract** class or implement a common **interface**).

The **Collection** interface defines common methods for querying (i.e., getting information from) and modifying (i.e., changing) the collection in some way. However, there are also various restrictions for each of the collection classes in terms of what they are allowed and not allowed to do when adding, removing and searching for data. We will look at the various classes that implement the **Collection** and **Map** interfaces.

It is not the purpose of this course to describe in-depth details on various kinds of collections and data structures. You will gain a deeper understanding of the advantages and disadvantages between data structures in your second year data structures course.

8.2 The List ADT

In real life, objects often appear in simple lists. For example, **Companies** may maintain a list of **Employees**, **Banks** may keep a list of **BankAccounts**, a **Person** may keep a list of "Things to Do". etc..

A **List** ADT allows us to access any of its elements at any time as well as insert or remove elements anywhere in the list at any time. The list will automatically shift the elements around in memory to make the needed room or reduce the unused space. The general **list** is the most flexible kind of list in terms of its capabilities.

> *A **list** is an abstract data type that implements an ordered collection of values, where the same value may occur more than once.*

We use a general **List** whenever we have elements coming in and being removed in a random order. For example, when we have a shelf of library books, we may need to remove a book from anywhere on the shelf and we may insert books back into their proper location when they are returned.

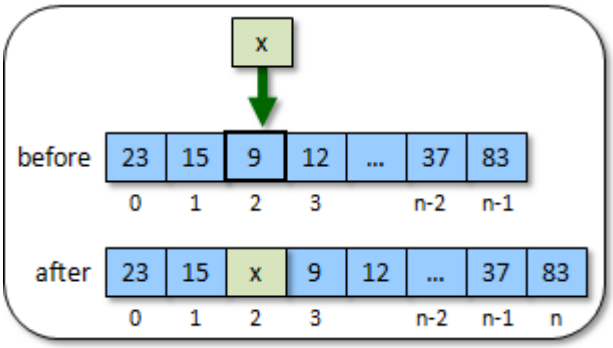
The elements in a general list are stored in a particular position in the list. As with arrays, elements in a general list are accessed according to their ***index*** position in the list.

The basic methods for inserting, removing, accessing and modifying items from a **List** are as follows:

add(**int** index, Object x)

Insert object **x** at position **index** in the list. Objects at positions **index + 1** through **n-1** move to the right by one position, increasing the size of the list by **1**.

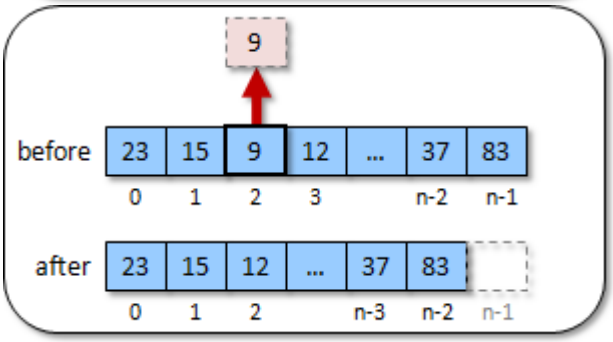
e.g., **aList.add(2, x)** will do this →



remove(**int** index)

Remove and return the object at position **index** in the list. Objects at positions **index + 1** through **n-1** move to the left by one position, decreasing the size of the list by **1**.

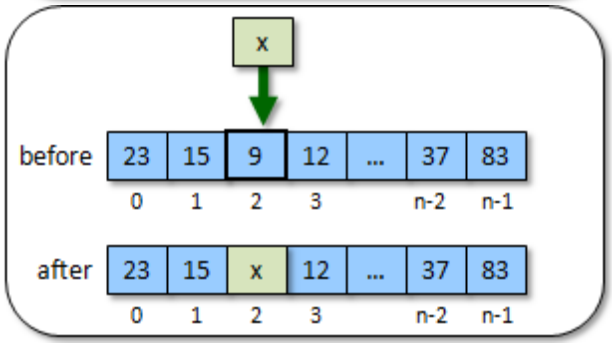
e.g., **aList.remove(2)** will return **9** →



set(**int** index, Object x)

Replace the object at position **index** in the list with the new object **x**. Objects at all other positions remain in their original position.

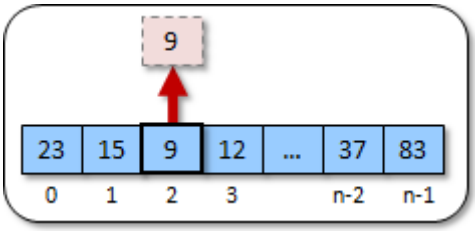
e.g., **aList.set(2, x)** will do this →



get(**int** index)

Return the object at position **index** in the list. The list is not changed in any way.

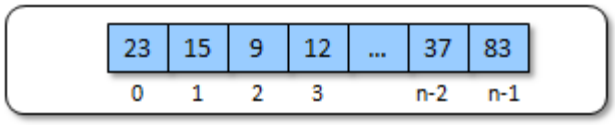
e.g., x = **aList.get(2)** will return **9** →



size()

Return the number of elements in the list.

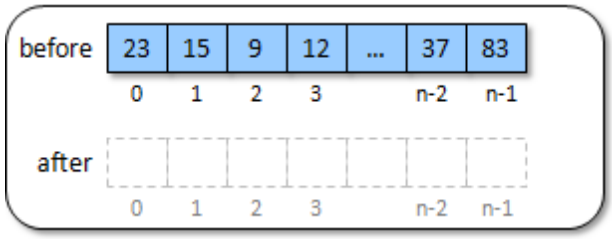
e.g., n = **aList.size()** will return **n** →



clear()

Remove all elements from the list.

e.g., **aList.clear()** will do this →

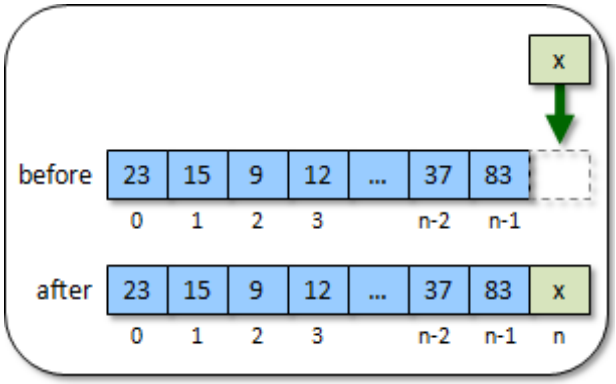


There are additional methods often available for convenience sake. Here are some:

add(Object x)

Insert object **x** at the end of the list, increasing the size of the list by **1**.

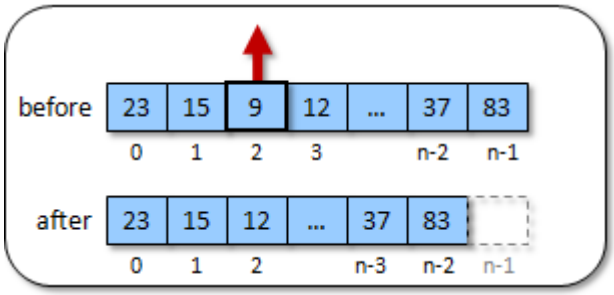
e.g., **aList.add(x)** will do this →



remove(Object x)

Remove the first occurrence of object **x** from the list. Assuming **x** was found at position **i**, then objects at positions **i + 1** through **n-1** move to the left by one position, decreasing the size of the list by **1**.

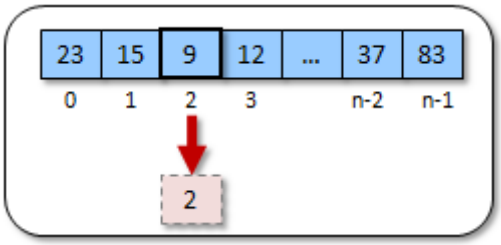
e.g., Integer **x = new** Integer(9);
aList.remove(x) will do this →



indexOf(Object x)

Return the position of the first occurrence of object **x** in the list.

e.g., Integer **x = new** Integer(9);
i = **aList.indexOf(x)** will return 2 →



isEmpty()

Return **true** if the number of elements in the list is **0**, otherwise return **false**.

does the same as this:

```
return (aList.size() == 0);
```

contains(Object x)

Return **true** if x is contained in the list, otherwise return **false**.

does the same as this:

```
for (int i=0; i<aList.size(); i++)
   if (aList.get(i).equals(x))
      return true;
return false;
```

In JAVA, the *List* ADT is called an **ArrayList** and it is located in the **java.util** package, which must be imported in order to use this data type.

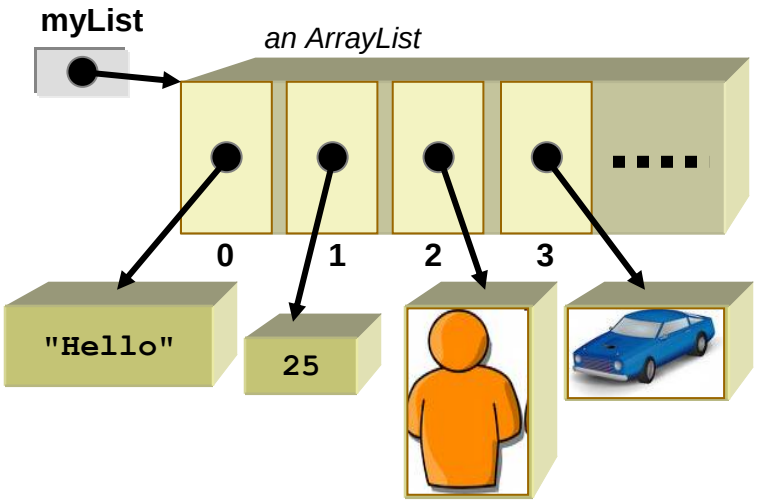
To create an **ArrayList**, we can simply call a constructor from the **ArrayList** class. Here is an example of creating an **ArrayList** and storing it in a variable so that we can use it:

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

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

```
import java.util.ArrayList;

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

        myList = new ArrayList();
        myList.add("Hello");
        myList.add(25);
        myList.add(new Person());
        myList.add(new Car());
        System.out.println(myList);
    }
}
```

Notice in the above code that we are adding a **String**, an **int**, a **Person** object and a **Car** 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 **Car** are defined classes that do not have **toString()** methods):

```
[Hello, 25, Person@addbf1, Car@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 **[ ]**.

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:

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

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:

```
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.

Here is an example that uses the **get()** and **size()** methods:

```
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
```

Since Lists are perhaps the most commonly used data structure in computer science, we will do a larger example so that we get a full understanding of how to use them properly.

Example:




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

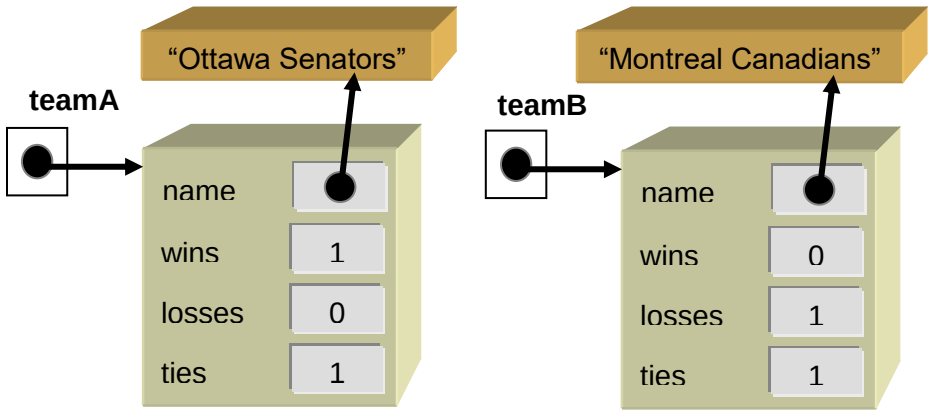
they played. Here is the basic class (review the previous chapters in the notes if any of this is not clear):

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

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

    // Get methods
    public String getName() { return name; }
    public int getWins() { return wins; }
    public int getLosses() { return losses; }
    public int getTies() { return ties; }

    // Modifying methods
    public void recordWin() { wins++; }
    public void recordLoss() { losses++; }
    public void recordTie() { ties++; }

    // Returns a text representation of a team
    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
    public int totalPoints() {
        return (this.wins * 2 + this.ties);
    }

    // Returns the total number of games played by the team
    public 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:

```
public 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.getName()+" just beat "+teamB.getName());
        teamA.recordWin();
        teamB.recordLoss();

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

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

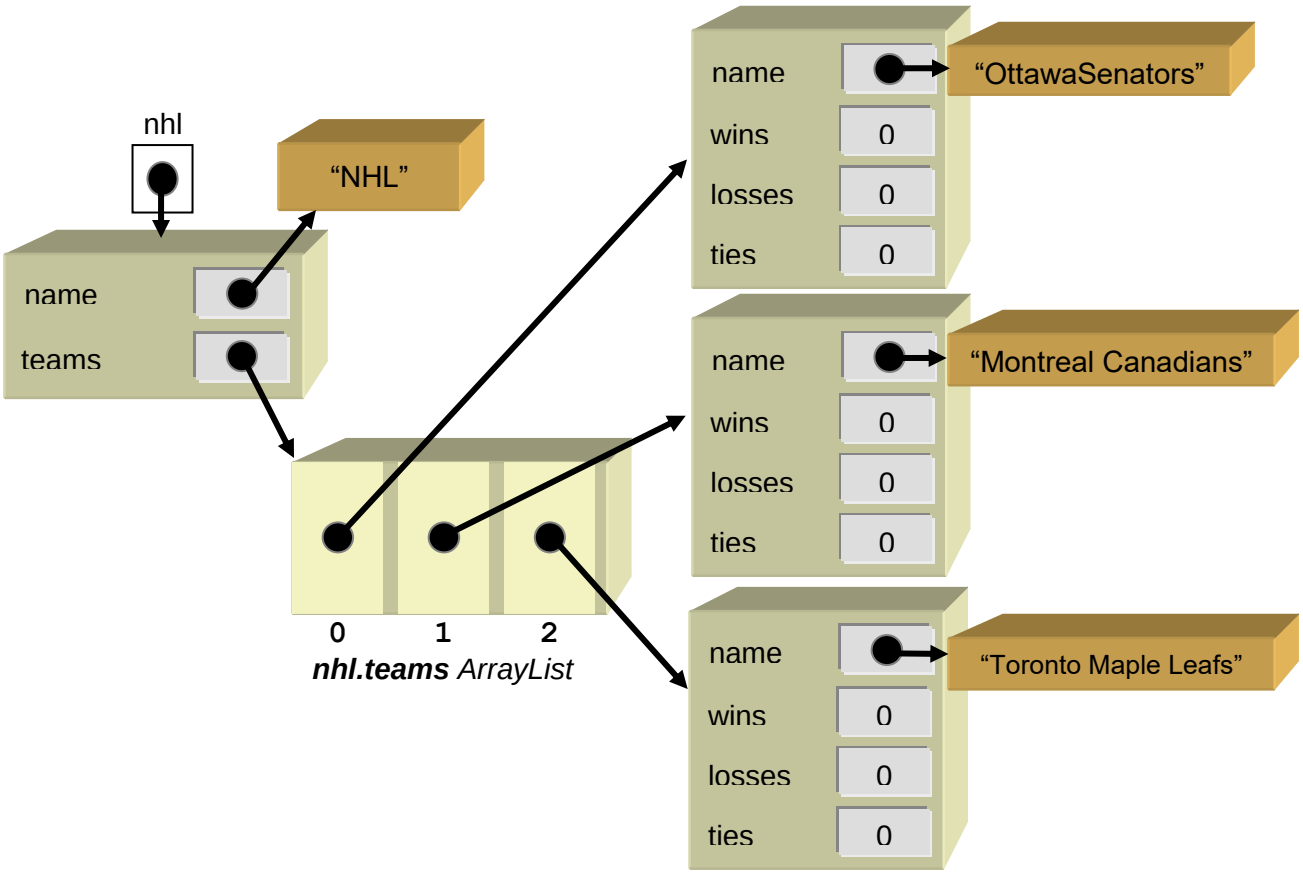
```
import java.util.ArrayList;

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

    public 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");
    }

    // Add the given team to the League
    public void addTeam(Team t) {
        teams.add(t);
    }
}
```

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:

```
League  nhl;

nhl = new League("NHL");
nhl.addTeam(new Team("Ottawa Senators"));
nhl.addTeam(new Team("Montreal Canadians"));
nhl.addTeam(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 **addTeam()** method which makes use of the **add()** method that is 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 **League** class called **showTeams()** to do this. The method will need to go through each team in the **teams** ArrayList and display the particular team's information … perhaps using the **toString()** method from the **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 **for** and **while** loops, but there is a special kind of **for** loop in JAVA that is to be used when traversing through a collection such as an **ArrayList**. This loop is called the "***for-each***" loop, and its structure is a little simpler than the traditional **for** loop. Below is how we can use the typical FOR loop as well as the "better" FOR-EACH loop to write the **showTeams()** method.

Using a Typical FOR Loop	Using a FOR-EACH Loop

`public void showTeams() {`
`  for (int i=0; i<teams.size(); i++) {`
`  System.out.println(teams.get(i));`
`  }`
`}`
	`public void showTeams() {`
	`  for (Team t: teams) {`
	`  System.out.println(t);`
	`  }`
	`}`

Notice that the ***for-each*** loop starts with **for** again, but this time the information within the parentheses **()** is different. The format of this information is as follows. First we specify the ***type*** of object that is in the ArrayList … in this case **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 **t**.

Then we use a colon **:** character followed by the name of the ArrayList that we want to loop through … in this case **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 **t** as we would use any other variable. In our example, **t** is the **Team** object that we are examining during that round through the loop. So **t** points to the 1st team in the league when we begin the loop, then it points to the 2nd team the next time through the loop, then the 3rd team etc..

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

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

        nhl = new League("NHL");

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

```
public void recordWinAndLoss(Team winner, Team loser) {
    winner.recordWin();
    loser.recordLoss();
}

public void recordTie(Team teamA, Team teamB) {
    teamA.recordTie();
    teamB.recordTie();
}
```

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.addTeam(team1 = new Team("Ottawa Senators"));
nhl.addTeam(team2 = new Team("Montreal Canadians"));
nhl.addTeam(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.addTeam(new Team("Ottawa Senators"));
nhl.addTeam(new Team("Montreal Canadians"));
nhl.addTeam(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):

```
public void recordWinAndLoss(String winnerName, String loserName) {
    //...
}

public 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:

```
private 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:

```
private Team teamWithName(String nameToLookFor) {
    Team   answer = null;
    for (Team t:   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 `nameToLookFor` 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.

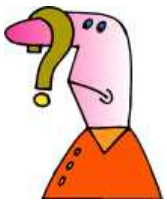
This method can be shortened as follows:

```
private Team teamWithName(String nameToLookFor) {
    for (Team t:   teams)
        if (t.getName().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:

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

    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    winner.recordWin();
    loser.recordLoss();
}

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

    teamA = this.teamWithName(teamAName);
    teamB = this.teamWithName(teamBName);
    teamA.recordTie();
    teamB.recordTie();
}
```

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.



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

    winner = this.teamWithName(winnerName);
    loser = this.teamWithName(loserName);
    if ((winner != null) && (loser != null)) {
        winner.recordWin();
        loser.recordLoss();
    }
}
```

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

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

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 a 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 into there instead as follows:

```
private Team teamWithName(String nameToLookFor) {
    for (Team t: teams)
        if (t.name.equals(nameToLookFor))
            return t;
    return null;
}
public void recordWinAndLoss(Team winner, Team loser) {
    if ((winner != null) && (loser != null)) {
        winner.recordWin();
        loser.recordLoss();
    }
}
public void recordTie(Team teamA, Team teamB) {
    if ((teamA != null) && (teamB != null)) {
        teamA.recordTie();
        teamB.recordTie();
    }
}
public void recordWinAndLoss(String winnerName, String loserName) {
    Team   winner, loser;

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

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

public 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:

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

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

```
public int totalGamesPlayed() {
    int  total = 0;
    for (Team t:   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:

```
public int totalGamesPlayed() {
    int  total = 0;
    for (Team t:   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:

```
public 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:

```
public Team firstPlaceTeam() {
    Team  teamWithMostPoints = null;

    for (Team t:   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:

```
public Team firstPlaceTeam() {
    int    points;
    Team   teamWithMostPoints = null;

    for (Team t:   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:

```
public Team firstPlaceTeam() {
    Team  teamWithMostPoints = null;

    for (Team t:   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 the 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:

```
public Team firstPlaceTeam() {
    Team  teamWithMostPoints = teams.get(0);

    for (Team t:   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 …

```
public Team firstPlaceTeam() {
    Team  teamWithMostPoints;

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

    teamWithMostPoints = teams.get(0);
    for (Team t:   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.

How could we write a method called **undefeatedTeams()** that returned an **ArrayList<Team>** of all teams that have never lost a game ? We would begin by specifying the proper return type:

```
public ArrayList<Team> undefeatedTeams() {
   ArrayList<Team>  undefeated = new ArrayList<Team>();

    for (Team t:   teams) {
        //...
    }
    return undefeated;
}
```

Now we would check each team that has not lost any games and add them to the result list:

```
public ArrayList<Team> undefeatedTeams() {
   ArrayList<Team>  undefeated = new ArrayList<Team>();

    for (Team t:   teams) {
        if (t.getLosses() == 0)
           undefeated.add(t);
    }
    return undefeated;
}
```

Another interesting method would be one that removes all teams from the league that have never won a game. Intuitively, here is what we may do:

```
public void removeLosingTeams() {
    for (Team t:   teams) {
        if (t.getWins() == 0)
            teams.remove(t);
    }
}
```

However, this code will not work since it will generate a **ConcurrentModificationException** in JAVA. That is, we need to be careful not to remove items from a list that we are iterating through. As it turns out, the FOR EACH loop does not allow us to remove while iterating through the list. Using a standard **FOR** loop, however, we can make it work. The following code "almost works" ... in that it does not produce an exception ... but something is still wrong.

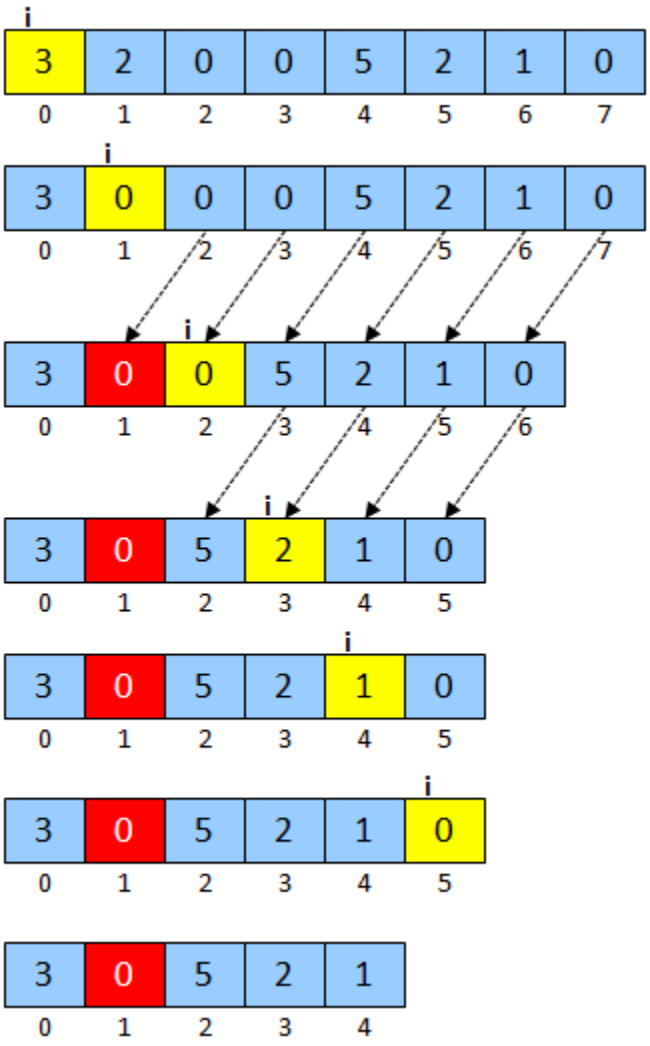
```
public void removeLosingTeams() {
    for (int i=0; i<teams.size(); i++) {
        Team t = teams.get(i);
        if (t.getWins() == 0)
            teams.remove(t);
    }
}
```

The code above may not remove all teams that have no wins. Why not ? Consider what happens if two teams in a row have no wins.

In the picture to the right, imagine that these values represent the number of wins for the teams in the **teams** list. Notice how the index **i** is moved along through the loop as shown by the yellow square. When the team at position 1 is encountered, it has no wins, so it is removed ... and all other teams are moved back one position in the list.

Then, we continue with the loop as usual. However, the next time through the loop the i has moved to position 2. However, a 0-win team (shown red) has just been shifted into position 1 but never checked. Thus, by the time the loop has ended we never checked the team in position 1 and therefore it remains in the list.

How can we fix this ? It is simple. Just ensure that we do not move the index to the next position in the case that we are doing a remove operation. We can accomplish this by



subtracting **1** from the index **i**, so that when the **for** loop increments **i**, it cancels out the increment and **i** remains in the same position.

```
public void removeLosingTeams() {
   for (int i=0; i<teams.size(); i++) {
      Team t = teams.get(i);
      if (t.getWins() == 0) {
         teams.remove(t);
         i--;
      }
   }
}
```

Now the code should work properly. Here is a program that can be used to test our methods:

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

        // Add a pile of teams to the league
        nhl.addTeam(new Team("Ottawa Senators"));
        nhl.addTeam(new Team("Montreal Canadians"));
        nhl.addTeam(new Team("Toronto Maple Leafs"));
        nhl.addTeam(new Team("Vancouver Cannucks"));
        nhl.addTeam(new Team("Edmonton Oilers"));
        nhl.addTeam(new Team("Washington Capitals"));
        nhl.addTeam(new Team("New Jersey Devils"));
        nhl.addTeam(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.recordTie("Detroit Red Wings", "New Jersey Devils");
        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("\nHere are the teams after recording the " +
                "wins, losses and ties:\n");
        nhl.showTeams();
```

```
        // Here are some statistics
        System.out.println("\nThe 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());
        System.out.println("The undefeated teams are " +
                nhl.undefeatedTeams());
        System.out.println("Removing teams that never won ... ");
        nhl.removeLosingTeams();
        System.out.println("The teams are: ");
        nhl.showTeams();
    }
}
```

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 Canadians 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.
The undefeated teams are [The Ottawa Senators have 2 wins, 0 losses and 1 ties.,
The Vancouver Cannucks have 1 wins, 0 losses and 0 ties.]
Removing teams that never won ...
The teams are:
The Ottawa Senators have 2 wins, 0 losses and 1 ties.
The Montreal Canadians 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 New Jersey Devils have 1 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.

8.3 The Queue ADT

Consider the **Queue** ADT.

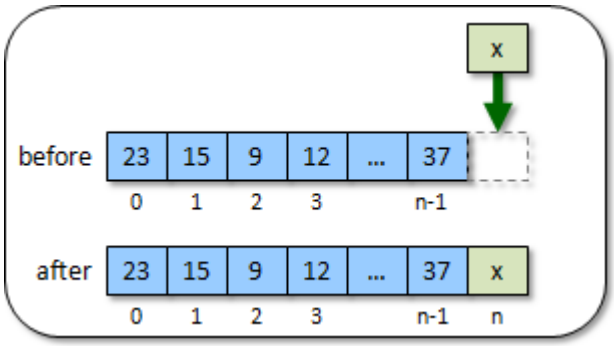
> *A **queue** is an abstract data type that stores elements in a first-in-first-out order. Elements are added at one end and removed from the other.*

Hence, the first element to be added to the queue is the first element to be taken out of the queue. This is analogous to a line-up that we see every day. The first person in line is the first person served (i.e., first-come-first-served). When people arrive, they go to the back of the line. People get served from the front of the line first. Therefore, with a queue, we **add to the back** and **remove from the front**. We are not allowed to insert or remove elements from the middle of the queue. Why is this restriction a good idea ? Well, depending on how the queue is implemented, it can be more efficient (i.e., faster) to insert and remove elements since we know that all such changes will occur at the front or back of the queue. Removing from the front may then simply require moving the "front-of-the-line pointer" instead of shifting elements over. Also, adding to the back may require extending the "back-of-the-line pointer". Typical methods for **Queues** are:

add(Object x)

Insert object **x** at the end of the queue. This operation is sometimes called **push(x)** or **enqueue(x)**.

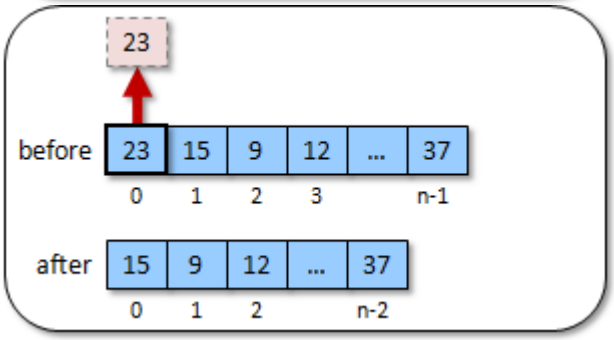
e.g., **aQueue.add(x)** will do this →



remove()

Remove and return the object at the front of the queue. The next item in the queue becomes the front item. This operation is sometimes called **pop()** or **dequeue()**.

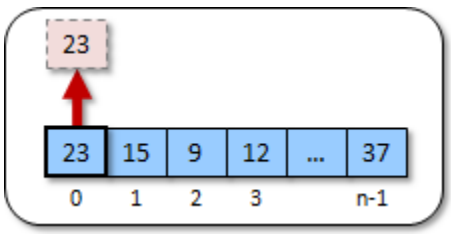
e.g., x = **aQueue.remove()** will return **23** →



peek()

Return (but do not remove) the object at the front of the queue. This operation is sometimes called **front()**.

e.g., x = **aQueue.peek()** will do this →



size(), isEmpty() and clear() Does the same as with lists

Another more specialized type of queue is the ***PriorityQueue***:

*A **PriorityQueue** is a queue in which the elements also maintain a priority.*

That is, we still add to the back of the queue and remove from the front, but elements with higher priority are automatically shifted closer to the front before lower priority elements.

As a real life example, when we go to the hospital for an "emergency", we wait in line (6 to 8 hours typically). We normally get served in the order that we came in at. However, if someone comes in after us who is bleeding or unconscious, they automatically get bumped up ahead of us since their injuries are likely more serious and demand immediate attention. We may think of a **PriorityQueue** as a ***sorted*** queue.

The **PriorityQueue** is used in the same way as a regular **Queue**, except that we must ensure that each element added to the queue is given a priority. Hence, we are sometimes required to specify the priority of an item when we add it to the queue:

add(**int** priority, Object x)

However, in JAVA, we simply include the priority of an object as part of the object itself, and so we will just use **add(Object x)** for priority queues in this course.

Example:

Consider the following **Person** class definition:

```
public class Person {
      protected String  name;
      protected int     age;

      public Person(String n, int a) {
            name = n;
            age = a;
      }

      public int getAge() { return age; }
      public String getName() { return name; }

      public String toString() {
            return age + " year old " + name;
      }
}
```

How could we simulate some people getting in a lineup and being served on a first-come-first-served basis ? We can use a **Queue** to do this. In JAVA, there are a few different

implementations of the **Queue** ADT. We will use the one called **ArrayBlockingQueue** which is in the **java.util.concurrent** package. Here is a simple test program that creates a lineup and then serves the people one at a time by taking the first person in line each time. Notice how the code is arranged so that a nice log output is obtained that we can follow along with to see if our program does what it is supposed to be doing.

```
import java.util.concurrent.ArrayBlockingQueue;

public class QueueTestProgram {
    public static void main(String[] args) {
        ArrayBlockingQueue<Person>    lineup;

        lineup = new ArrayBlockingQueue<Person>(10);

        System.out.print("Adding three customers to the line ... ");
        lineup.add(new Person("Bob", 12));
        lineup.add(new Person("Mary", 6));
        lineup.add(new Person("Steve", 10));

        System.out.println("Here is who is in line at the moment:");
        System.out.println(lineup);

        System.out.print("Serving next customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Serving another customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Adding three more customers to the line ... ");
        lineup.add(new Person("Ralph", 16));
        lineup.add(new Person("Jen", 13));
        lineup.add(new Person("Max", 18));
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Serving next customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);

        System.out.print("Adding four customers to the line ... ");
        lineup.add(new Person("Dave", 4));
        lineup.add(new Person("Sam", 17));
        lineup.add(new Person("Lyn", 8));
        lineup.add(new Person("Betty", 9));
        System.out.println("Here is who is in line now:");
        System.out.println(lineup);
        System.out.print("Here is who is at the front of the line ...");
        System.out.println(lineup.peek());
        System.out.print("Serving next customer ... ");
        System.out.println(lineup.remove());
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
        System.out.println("Serving all remaining customers ... ");
```

```
        while(!lineup.isEmpty()) {
            System.out.print("Serving next customer ... ");
            System.out.println(lineup.remove());
        }
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
    }
}
```

Here is the output that is produced:

```
Adding three customers to the line ... Here is who is in line at the moment:
[12 year old Bob, 6 year old Mary, 10 year old Steve]
Serving next customer ... 12 year old Bob
Here is who is in line now:
[6 year old Mary, 10 year old Steve]
Serving another customer ... 6 year old Mary
Here is who is in line now:
[10 year old Steve]
Adding three more customers to the line ... Here is who is in line now:
[10 year old Steve, 16 year old Ralph, 13 year old Jen, 18 year old Max]
Serving next customer ... 10 year old Steve
Here is who is in line now:
[16 year old Ralph, 13 year old Jen, 18 year old Max]
Adding four customers to the line ... Here is who is in line now:
[16 year old Ralph, 13 year old Jen, 18 year old Max, 4 year old Dave, 17
year old Sam, 8 year old Lyn, 9 year old Betty]
Here is who is at the front of the line ...16 year old Ralph
Serving next customer ... 16 year old Ralph
Here is who remains in the line:
[13 year old Jen, 18 year old Max, 4 year old Dave, 17 year old Sam, 8 year
old Lyn, 9 year old Betty]
Serving all remaining customers ...
Serving next customer ... 13 year old Jen
Serving next customer ... 18 year old Max
Serving next customer ... 4 year old Dave
Serving next customer ... 17 year old Sam
Serving next customer ... 8 year old Lyn
Serving next customer ... 9 year old Betty
Here is who remains in the line:
[]
```

Example:

In a **PriorityQueue**, when we add items, they usually get shuffled around inside according to their priority. Therefore, we may not necessarily know the order of the items afterwards … except that they will be in some sort of prioritized order.

Consider the **QueueTestProgram**. We can change the **ArrayBlockingQueue** to **PriorityQueue** to have a prioritized queue for our people. However, if we were to run the code, we would get an exception:

```
java.lang.ClassCastException: Person cannot be cast to java.lang.Comparable
```

The problem is that JAVA does not know how to compare **Person** objects in order to be able to sort them. It is telling us that **Person** must implement the **Comparable** interface. Instead of supplying a priority when we add the objects to the queue, the items are sorted by means of a **Comparable** interface. That means, each object that we store in the **PriorityQueue**, must implement methods **compareTo()** … which are used determine the sort order (i.e., priority).

So, we should add **implements Comparable<Person>** to the **Person** class definition:

```
public class Person implements Comparable<Person> {
    ...
}
```

Interestingly, the additional **<Person>** at the end of **Comparable** indicates to JAVA that we will only be comparing **Person** objects, not **Person** objects with other types of objects.

But how do we write a **compareTo()** method ? It takes a single object parameter:

```
public int compareTo(Person  p) { ... }
```

The method returns an **int**. This integer reflects the ordering between the receiver and the parameter. If a negative value is returned from the method, this informs JAVA that the receiver has higher priority (i.e., comes before in the ordering) than the incoming parameter object. Likewise, a positive value indicates lower priority and a zero value indicates that they are equal priority.

Let's now give it a try for **Person** objects. If we want to prioritize by means of their increasing ages (i.e., younger first), this would be the **compareTo()** method:

```
public int compareTo(Person  p) {
    return (this.age - p.age);
}
```

Assume now that we ran the following program:

```
import java.util.PriorityQueue;

public class PriorityQueueTestProgram {
    public static void main(String[] args) {
        PriorityQueue<Person>  lineup;
        lineup = new PriorityQueue<Person>(10);    // at most 10 allowed in line

        System.out.print("Adding 10 customers to the line ... ");
        lineup.add(new Person("Bob", 12));
        lineup.add(new Person("Mary", 6));
        lineup.add(new Person("Steve", 10));
        lineup.add(new Person("Ralph", 16));
        lineup.add(new Person("Jen", 13));
        lineup.add(new Person("Max", 18));
        lineup.add(new Person("Dave", 4));
        lineup.add(new Person("Sam", 17));
        lineup.add(new Person("Lyn", 8));
```

```
        lineup.add(new Person("Betty", 9));

        System.out.println("Here is who is in line now:");
        System.out.println(lineup);
        System.out.print("Here is who is at the front of the line ...");
        System.out.println(lineup.peek());

        System.out.println("Serving all customers ... ");
        while(!lineup.isEmpty()) {
            System.out.print("Serving next customer ... ");
            System.out.println(lineup.remove());
        }
        System.out.println("Here is who remains in the line:");
        System.out.println(lineup);
    }
}
```

Interestingly, the output is as follows:

```
Adding 10 customers to the line ... Here is who is in line now:
[4 year old Dave, 8 year old Lyn, 6 year old Mary, 12 year old Bob, 9 year
old Betty, 18 year old Max, 10 year old Steve, 17 year old Sam, 16 year old
Ralph, 13 year old Jen]
Here is who is at the front of the line ...4 year old Dave
Serving all customers ...
Serving next customer ... 4 year old Dave
Serving next customer ... 6 year old Mary
Serving next customer ... 8 year old Lyn
Serving next customer ... 9 year old Betty
Serving next customer ... 10 year old Steve
Serving next customer ... 12 year old Bob
Serving next customer ... 13 year old Jen
Serving next customer ... 16 year old Ralph
Serving next customer ... 17 year old Sam
Serving next customer ... 18 year old Max
Here is who remains in the line:
[]
```

Notice that the items in the queue do not seem sorted at all ! That is because a **PriorityQueue** does not actually sort the items, it simple makes sure that the item at the front of the queue is the one with highest priority. In this case, that is the youngest person … which is indeed at the front of the queue. To get the items in sorted order, we simply extract them from the queue one at a time as shown in the **while** loop from the code above. Indeed, you can see that as we extract the items one at a time, they come out in properly prioritized order.

What if we wanted to prioritize the people by their last names instead ? To do this, we would need to alter the **compareTo()** method to compare names, not ages like this:

```
public int compareTo(Person  p) {
    return (this.name.compareTo(p.name));
}
```

Notice that there is a **compareTo()** method in the String class. This compares two strings alphabetically, making sure that the first one in alphabetical order has higher priority.

What if we want to be able to sort multiple different ways in our program ? That is, sometimes we might want to sort increasing by age, decreasing by age, alphabetical, by phone number, etc.. To do this, we can define some static constants in the **Person** class ... one for each kind of sorting strategy ... and then create a static/class variable that indicates which sort strategy that we want to use at any time. Then, in the **compareTo()** method, we could simply check that static variable and make our decision as to how to sort. Here is the code:

```
public class Person implements Comparable<Person> {
    public static final byte    ALPHABETICAL = 0;
    public static final byte    INCREASING_AGE = 1;
    public static final byte    DECREASING_AGE = 2;

    public static byte          SortStrategy = INCREASING_AGE;

    protected String   name;
    protected int      age;

    public Person(String n, int a) {
        name = n;
        age = a;
    }

    public int compareTo(Person p) {
        switch(SortStrategy) {
            case ALPHABETICAL: return name.compareTo(p.name);
            case INCREASING_AGE: return age - p.age;
            case DECREASING_AGE: return p.age - age;
        }
        return 0;
    }

    public int getAge() { return age; }
    public String getName() { return name; }

    public String toString() {
        return age + " year old " + name;
    }
}
```

Then to test it out, we simply set the **SortStrategy** at any time:

```
import java.util.PriorityQueue;
public class PriorityQueueTestProgram2 {
    public static void main(String[] args) {
        PriorityQueue<Person>  lineup = new PriorityQueue<Person>(10);

        System.out.println("Adding 10 customers to the line ... ");
        lineup.add(new Person("Bob", 12));
        lineup.add(new Person("Mary", 6));
        lineup.add(new Person("Steve", 10));
        lineup.add(new Person("Ralph", 16));
        lineup.add(new Person("Jen", 13));
        lineup.add(new Person("Max", 18));
        lineup.add(new Person("Dave", 4));
        lineup.add(new Person("Sam", 17));
        lineup.add(new Person("Lyn", 8));
        lineup.add(new Person("Betty", 9));
```

```
        // Set priority to be alphabetically
        Person.SortStrategy = Person.ALPHABETICAL;
        // Since items are sorted when added, we need to re-add every item again:
        ArrayList<Person>  items = new ArrayList<Person>(lineup);
        lineup.clear();
        lineup.addAll(items);

        System.out.println("Serving all customers ... ");
        while(!lineup.isEmpty()) {
            System.out.print("Serving next customer ... ");
            System.out.println(lineup.remove());
        }
    }
}
```

Here is the output ... notice how the people are removed in alphabetical order of their name:

```
Adding 10 customers to the line ...
Serving all customers ...
Serving next customer ... 9 year old Betty
Serving next customer ... 12 year old Bob
Serving next customer ... 4 year old Dave
Serving next customer ... 13 year old Jen
Serving next customer ... 8 year old Lyn
Serving next customer ... 6 year old Mary
Serving next customer ... 18 year old Max
Serving next customer ... 16 year old Ralph
Serving next customer ... 17 year old Sam
Serving next customer ... 10 year old Steve
```

8.4 The Deque ADT

Consider the **Deque** ADT:

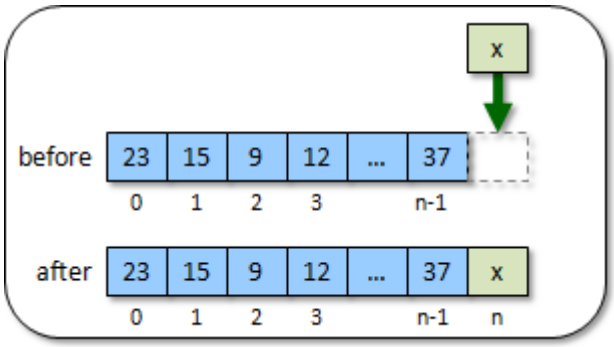
> *A **deque** is an abstract data type that is analogous to a **d**ouble-**e**nded **que**ue. Elements are added/removed to/from either end.*

A deque allows us to add/remove from the front or the back of the queue at any time, but modifications to the middle are not allowed. It has the same advantages of a regular single-ended queue, but is a little more flexible in that it allows removal from the back of the queue and insertion at the front. An example of where we might use a deque is when we implement "Undo" operations in a piece of software. Each time we do an operation, we **add** it to the front of the deque. When we do an undo, we **remove** it from the front of the deque. Since undo operations usually have a fixed limit defined somewhere in the options (i.e., maximum 20 levels of undo), we remove from the back of the deque when the limit is reached. Typical methods for **Deques** are:

addFirst(Object x) and addLast(Object x)

Insert object **x** at the front (or back) of the deque.

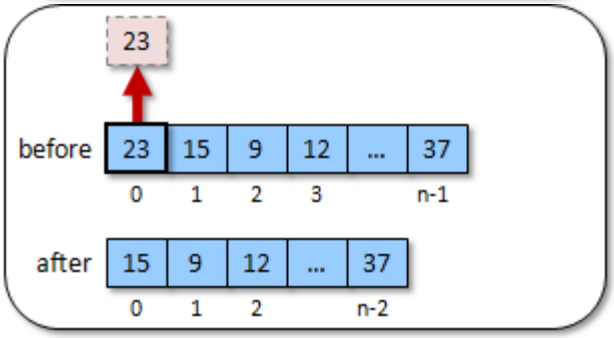
e.g., **aDeque.addLast(x)** will do this →



removeFirst() and removeLast()

Remove and return the object at the front (or back) of the deque. The next item in the deque becomes the front (or back) item.

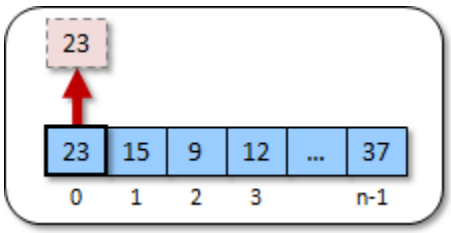
e.g., x = **aDeque.removeFirst()** will return **23** →



peekFirst() and peekLast()

Return (but do not remove) the object at the front (or back) of the deque.

e.g., x = **aDeque.peekFirst()** will do this →



size(), isEmpty() and clear()

Does the same as with queues

Example:

Consider simulating an "undo list". Many applications allow the user to perform various operations and then select **undo** to undo a mistake that was made. Often, programs will allow you to select **undo** many times to undo many previous operations in sequence. However, most programs have a limit of how many times you can undo (e.g., at most 20 undo operations). After the limit is reached, it is impossible to undo the older operations.

In JAVA, we can use an **ArrayDeque** (in the **java.util** package) to maintain our **undo** deque. Each time the user performs an operation, it is added to the end of the deque. Hence the end of the deque has the most recent operation performed while the front of the deque has the oldest operation performed.

Assuming that we were to set a limit of 5 **undo** operations. The undo deque will hold at most 5 operations. When a 6th **undo** operation is performed ... what do we do ? We need to add it to the end of the deque, but then remove the oldest undo operation from the front of the deque.

Here is an example of how we could do this. The example here shows how simple operations (in the form of Strings) can be maintained in an **ArrayDeque** ADT:

```
import java.util.ArrayDeque;

public class DequeTestProgram {
    private static int UNDO_LIST_CAPACITY = 5;

    private static ArrayDeque<String>  operations;

    private static void performOperation(String x) {
        if (operations.size() == UNDO_LIST_CAPACITY)
            operations.removeFirst();
        operations.addLast(x);
    }
    private static void undo() {
        operations.removeLast();
    }

    public static void main(String[] args) {
        operations = new ArrayDeque<String>();
        System.out.println("Simulating some cut/paste/move operations ... ");
        performOperation("cut1");
        performOperation("paste1");
        performOperation("move1");
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating an undo operation ...");
        undo();
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating another undo operation ...");
        undo();
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating some more cut/paste/move operations ... ");
        performOperation("cut2");
        performOperation("paste2");
        performOperation("move2");
        performOperation("move3");
        System.out.println("Here is the undo list now: " + operations);

        System.out.println("Simulating some more paste operations ... ");
        performOperation("paste3");
        performOperation("paste4");

        System.out.println("Here is the undo list now: " + operations);
    }
}
```

Here is the output ... as you can see, the deque is maintained with a size of at most 5:

```
Simulating some cut/paste/move operations ...
Here is the undo list now: [cut1, paste1, move1]
Simulating an undo operation ...
Here is the undo list now: [cut1, paste1]
Simulating another undo operation ...
Here is the undo list now: [cut1]
Simulating some more cut/paste/move operations ...
Here is the undo list now: [cut1, cut2, paste2, move2, move3]
Simulating some more paste operations ...
Here is the undo list now: [paste2, move2, move3, paste3, paste4]
```

8.5 The Stack ADT

Consider the **Stack** ADT:

> *A **stack** is an abstract data type that stores elements in a last-in-first-out (LIFO) order. Elements are added and removed to/from the top only.*

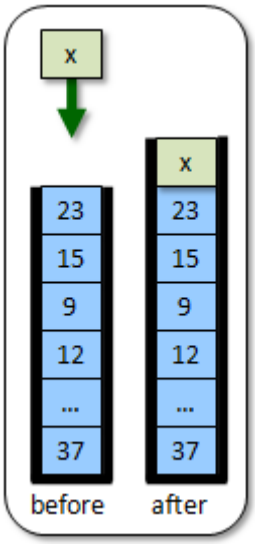
A stack stores items one on top of another. When a new item comes in, we place it on the top of the stack and when we want to remove an item, we take the top one from the stack. Stacks are used for many applications in computer science such as syntax parsing, memory management, reversing data, backtracking, etc..

Typical methods for **Stacks** are:

push(Object x)

Insert object **x** at the top of the stack.

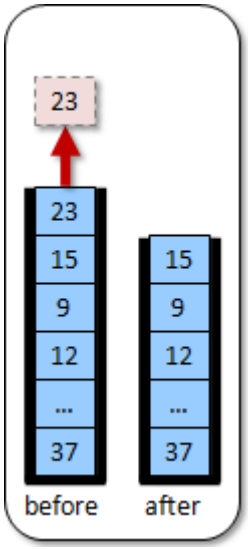
e.g., **aStack.push(x)**



pop()

Remove and return the object at the top of the stack.

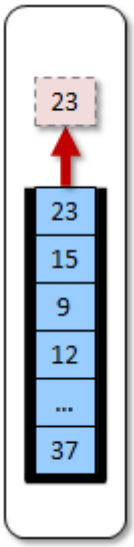
e.g., x = **aStack.pop()**



peek()

Return (but do not remove) the object at the top of the stack.

e.g., x = **aStack.peek()**

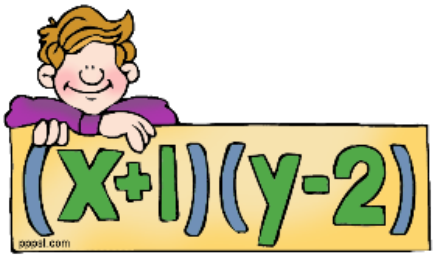


size(), isEmpty() and clear()

Does the same as lists

Example:

Consider a math expression that contains numbers, operators and parentheses (i.e., round brackets). How could we write a program that takes a **String** representing a math expression and then determines whether or not the brackets match properly (i.e., each opening bracket has a matching closing bracket in the right order) ?




```
"((23 + 4 * 5) - 34) + (34 - 5))"  // no match
"((23 + 4 * 5) - 34) + ((34 - 5)"  // no match
"((23 + 4 * 5) - 34) + (34 - 5)"   // match
```

How would we approach solving this problem? Well, we need to understand the process. I'm sure that you realize that we need to look at all the String's characters. Perhaps from start to end with a loop, but then what do we do ?

Let's assume that we are not interested in determining whether the formula makes sense but rather that each opening bracket is matched by a closing bracket. Therefore, we are interested in the bracket characters **(** and **)**, but not the other characters. When encountering an open bracket as we go through the characters of the string, we need to do something. We might think right away of trying to find the matching closing bracket for each open bracket, but that is not as easy as it sounds. There are many special cases that can be tricky.

A simpler approach would be to make sure that whenever we find a closing bracket, we just need to make sure that we already encountered an open bracket to match with it. This can be done by keeping a count of the number of open brackets. When encountering an opening bracket we increment the counter and when encountering a closing bracket we decrement the counter. If, when all done, the counter is not zero, there is no match. Otherwise the brackets match. Consider these cases:

```
"()"            // counter = 0, match
"() ("          // counter = 1, no match
"(((("          // counter = 3, no match
"((()) ())"     // counter = 0, match
"(())))"        // counter = -1, no match
""              // counter = 0, match
```

There is a special case that we did not consider. If the counter ever becomes negative before we are done, then we must have encountered a closing bracket before an open bracket … and there is no match:

```
")("       // counter = -1, no match
"())("     // counter = -1, no match
```

So, how do we write the code ? We can use a **FOR** loop and some **IF** statements to check for brackets as follows:

```
public static boolean bracketsMatch(String s) {
    int   count = 0;
    char  c;

    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);
        if (c == '(') count++;
        if (c == ')') count--;
        if (count < 0)
            return false;
    }
    return count == 0;
}
```

Here is a test program to try it out:

```
import java.util.*;
public class BracketMatchTestProgram {
    public static boolean bracketsMatch(String s) { /* code as above */ }

    public static void main(String[] args) {
        Scanner    keyboard = new Scanner(System.in);
        String     aString;

        do {
            System.out.println("Please enter expression: (<cr> to quit)");
            aString = keyboard.nextLine();

            if (bracketsMatch(aString))
                System.out.println("The brackets match");
            else
                System.out.println("The brackets do not match");
        } while (aString.length() > 0);
    }
}
```

Here are some testing results:

```
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5))
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + ((34 - 5)
The brackets do not match
Please enter expression: (<cr> to quit)
((23 + 4 * 5) - 34) + (34 - 5)
The brackets match
Please enter expression: (<cr> to quit)
()
The brackets match
Please enter expression: (<cr> to quit)
)(
```

```
The brackets do not match
Please enter expression: (<cr> to quit)
(((
The brackets do not match
Please enter expression: (<cr> to quit)
((())())
The brackets match
Please enter expression: (<cr> to quit)
((()))
The brackets do not match
Please enter expression: (<cr> to quit)
)(
The brackets do not match
Please enter expression: (<cr> to quit)

The brackets match
```

I think that our solution works fine. The bracket-matching example above is not very difficult, but what if we have 3 kinds of brackets **(**, **[**, and **{** ? Consider this example:

```
"{2 +(3 *[4 - 5})]"   // not supposed to match
```

Maybe we can keep 3 counters ? If we just keep three counters separately, we cannot tell whether the brackets are well-formed with respect to one another (e.g., closing round bracket for opening round bracket). We somehow need to know the ordering of each type of bracket so that we can ensure the reverse ordering when we find the closing brackets.

The need for backtracking may seem a little clearer if we consider a different application of the bracket matching program. Suppose that we want to match the brackets in our JAVA code…

```
public class PrintWriterTestProgram {
      public static void main(String[] args) {
            try {
                  BankAccount  aBankAccount;
                  PrintWriter  out;

                  aBankAccount = new BankAccount("Rob Banks");
                  aBankAccount.deposit(100);
                  out = new PrintWriter(new FileWriter("myAccount2.dat"));
                  out.println(aBankAccount.getOwner());
                  out.println(aBankAccount.getAccountNumber());
                  out.println(aBankAccount.getBalance());
                  out.close();
            } catch (FileNotFoundException e) {
                  System.out.println("Error: Cannot open file for writing");
            } catch (IOException e) {
                  System.out.println("Error: Cannot write to file");
            }
      }
}
```

Here we see, for example, that the portion of code inside the **class** definition must have all of its brackets matching, and that involves matching the code inside the **main** method's body and

then within the **try** block etc… The compiler does this kind of bracket-matching to make sure that your code is well-formed.

The **stack** data structure is designed for this purpose. It allows us to back-track … which is essentially what we need to do when finding a closing bracket. Here is how we can use a stack. When we find an open bracket, we put it on the top of the stack, regardless of its type. When we find a closing bracket, we take the top opening bracket from the stack and check to see if it is the same type as the closing bracket. If not, the brackets are in the wrong order, otherwise all is fine and we continue onwards. If, when encountering a closing bracket, we find that the stack is empty, then there is no match either.

Let's look at the code. In JAVA, we make a **Stack** by simply calling its constructor:

```
Stack  aStack = new Stack();
```

However, in our case, we are going to be placing bracket characters on the stack. Therefore we should specify this as follows:

```
Stack<Character>  aStack = new Stack<Character>();
```

Then, we need to use the appropriate **Stack** methods: Here is the resulting code:

```
public static boolean bracketsMatch2(String s) {
    Stack<Character>   aStack;
    char               c;

    aStack = new Stack<Character>();
    for (int i=0; i<s.length(); i++) {
        c = s.charAt(i);

        if ((c == '(') || (c == '[') || (c == '{'))  // got open bracket
            aStack.push(c);

        if ((c == ')') || (c == ']') || (c == '}')) { // got closed bracket
            if (aStack.isEmpty())
                return false;    // no open bracket for this closed one

            char top = aStack.pop();  // get the last opening bracket found
            if (((c == ')') && (top != '(')) ||
                ((c == ']') && (top != '[')) ||
                ((c == '}') && (top != '{')))
                return false;  // wrong closing bracket for last opened one
        }
    }
    return aStack.isEmpty();  // No match if brackets are left over
}
```

Notice in the above code that it never has **return true** anywhere. In fact, it is only at the very end, once we have checked all characters that there is a chance for the method to return **true**. This will happen if the stack is empty (i.e., all open brackets have been matched with closing ones). If desired, you can simplify the above code by replacing the **IF** statements with a **SWITCH** statement as follows:

```
switch(c) {
    case '(':
    case '[':
    case '{':
        aStack.push(c); // got open bracket
        break;
    case ')':
        if (aStack.isEmpty() || (aStack.pop() != '('))
            return false;
        break;
    case ']':
        if (aStack.isEmpty() || (aStack.pop() != '['))
            return false;
        break;
    case '}':
        if (aStack.isEmpty() || (aStack.pop() != '{'))
            return false;
        break;
}
```

Here are some testing results that we would obtain if we replaced our previous **bracketsMatch()** method with this new method:

```
Please enter the expression: (just <cr> to quit)
](){}[
The brackets do not match
Please enter the expression: (just <cr> to quit)
()[]{}
The brackets match
Please enter the expression: (just <cr> to quit)
{{{{[[]]}}}}
The brackets match
Please enter the expression: (just <cr> to quit)
{{{{{{
The brackets do not match
Please enter the expression: (just <cr> to quit)
}}}}}}
The brackets do not match
Please enter the expression: (just <cr> to quit)
(((()[]{})[()[]{])
The brackets do not match
Please enter the expression: (just <cr> to quit)
The brackets match
```

Challenge: Could you adjust the code above to read in a JAVA file instead of using a fixed string and have it ensure that the brackets match ? Chapter 11 discusses file I/O.

8.6 The Set ADT

Consider the **Set** ADT:

> *A **set** is an abstract data type that does not allow duplicate elements to be added.*

That is, there cannot be two elements e_1 and e_2 such that e_1**.equals(**e_2**)**. Any attempt to add duplicate elements is ignored. Sets differ from Lists in that the elements are not kept in the same order as when they were added. Sets are generally unordered, which means that the particular location of an element may change according to the particular set implementation.

Typical operations for **Sets** are:

- add(Object x)
- remove(Object x)

These operations work similar to that of **Lists** with the exception that the elements are not necessarily maintained in the same order that they were added in. Also, items are not guaranteed to be added to the **Set** because the **add()** operation will not allow any duplicate items to be added.

Example:

Suppose we wanted to maintain a list of DVD titles in our personal movie collection. Likely we do not want to have two or more of the same DVD. We can use a **Set** to avoid duplicates. In JAVA, we use the **HashSet** class which is in the **java.util** package. Consider a simple **Movie** class as follows:

```
public class Movie {
    private String     title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }

    public String toString() {
        return "Movie: \"" + title + "\"";
    }
}
```

Now consider the following code which simulates some inventory at a dvd-rental store. The code makes use of the simple **Movie** object by adding **10** movies from among **5** unique titles … hence many duplicates. The code makes us of **Math.random()** so that the inventory is different each time we run the program.

```
import java.util.*;
public class SetTestProgram1 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                new Movie("Monsters Vs. Aliens"),
                new Movie("Marley & Me"),
                new Movie("Hotel For Dogs"),
                new Movie("The Day the Earth Stood Still")};
        ArrayList<Movie>   inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int)(Math.random()*5)]);
        }

        for (Movie m:  inventory)
            System.out.println(m);
    }
}
```

Of course, each time that we run this code the result is different. Here is an example of what we may see:

```
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Bolt"
Movie: "Bolt"
```

Can we adjust the **FOR** loop so that it only displays unique movies ? No. We would have to do some extra work of making a new list with the duplicates removed. So, we could replace the **FOR** loop with the following:

```
ArrayList<Movie>  uniqueList = new ArrayList<Movie>();

for (Movie m:  inventory) {
   if (!uniqueList.contains(m))
      uniqueList.add(m);
}
for (Movie m:  uniqueList) {
   System.out.println(m);
}
```

This would produce the following output (according to the earlier results):

```
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
```

However, there is an easier way to do this. Consider what happens if we change the inventory from an **ArrayList** to a **HashSet** as follows:

```
HashSet<Movie>   inventory = new HashSet<Movie>();
```

Our code would produce the following output (which varies according to the randomness):

```
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
```

Notice that the duplicates were removed. The **HashSet** prevented any duplicates from being added. Therefore, we have lost all duplicate copies from our inventory, which can be bad. Perhaps it would be better to only use a **HashSet** when displaying the inventory, so that we don't destroy the duplicate movies. This is easily done by creating an extra **HashSet** variable (**displayList** in this case) and using the **HashSet** constructor that takes a **Collection** parameter:

```
import java.util.*;

public class SetTestProgram2 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                new Movie("Monsters Vs. Aliens"),
                new Movie("Marley & Me"),
                new Movie("Hotel For Dogs"),
                new Movie("The Day the Earth Stood Still")};

        ArrayList<Movie>   inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int)(Math.random()*5)]);
        }

        System.out.println("Here are the unique movies:");
        HashSet<Movie>   displayList = new HashSet<Movie>(inventory);
        for (Movie m:  displayList)
            System.out.println(m);

        System.out.println("\nHere is the whole inventory:");
        for (Movie m:  inventory)
            System.out.println(m);
    }
}
```

Notice the parameter to the **HashSet** constructor. This constructor will ensure to add all the elements from the inventory collection to the newly create **HashSet**. Then, in the **FOR** loop, we use this new **HashSet** for display purposes, while the original inventory remains unaltered. Here is the output:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
```

```
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"

Here is the whole inventory:
Movie: "Bolt"
Movie: "Bolt"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "Hotel For Dogs"
Movie: "Bolt"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
```

So, it is easy to remove duplicates from any collection … we simply create a new **HashSet** from the collection and it removes the duplicates for us.

However, there is one potential problem with our code. In the above code, the duplicate movies all represented the same exact object in memory … that is … all duplicates were identical to one another. However, it is more common to have two equal movies which are not identical. So, consider this code … notice the equal (but not identical) **Movie** objects:

```
import java.util.*;

public class SetTestProgram3 {
    public static void main(String[] args) {
        Movie[] dvds = {new Movie("Bolt"),
                new Movie("Monsters Vs. Aliens"),
                new Movie("Marley & Me"),
                new Movie("Monsters Vs. Aliens"),
                new Movie("Hotel For Dogs"),
                new Movie("Hotel For Dogs"),
                new Movie("Monsters Vs. Aliens"),
                new Movie("The Day the Earth Stood Still"),
                new Movie("The Day the Earth Stood Still"),
                new Movie("The Day the Earth Stood Still"),
                new Movie("The Day the Earth Stood Still")};

        ArrayList<Movie>   inventory = new ArrayList<Movie>();

        // Add 10 random movies from the list of dvds
        for (int i=0; i<10; i++) {
            inventory.add(dvds[(int)(Math.random()*11)]);
        }

        System.out.println("Here are the unique movies:");
        HashSet<Movie>   displayList = new HashSet<Movie>(inventory);
        for (Movie m:  displayList)
            System.out.println(m);

        System.out.println("\nHere is the whole inventory:");
        for (Movie m:  inventory)
            System.out.println(m);
    }
}
```

Here is the result:

```
Here are the unique movies:
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Bolt"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Monsters Vs. Aliens"

Here is the whole inventory:
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"
Movie: "Monsters Vs. Aliens"
Movie: "Bolt"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
```

Notice that there are many duplicates still in the **Set**. The problem is that the **Set** classes make use of a particular method in order to determine whether or not an object is the same as another one already in the set. The method used by java to determine equality between objects is called the **equals(Object x)** method. So, in general, the **equals()** method can be used to compare any two objects to determine whether or not they are the same:

```
anObject.equals(anotherObject)    // returns either true or false
```

All objects in JAVA have a default inherited **equals()** method which resides in the **Object** class. Unfortunately however, this default **equals()** method does the following:

```
public boolean equals(Object x) {
      return (this == x);
}
```

Therefore, the method, by default, will only return **true** if the object is the same exact object as **x** (i.e., the same memory location reference). Therefore, when we compare two **Movie** objects that were created using their own constructors, these movies can never be equal by default since they each reside in their own individual memory locations.

```
Movie m1 = new Movie("Monsters Vs. Aliens");
Movie m2 = new Movie("Monsters Vs. Aliens");
Movie m3 = m2;
m1.equals(m2);    // returns false
m2.equals(m3);    // returns true
```

In order to avoid this problem, we need to re-define the **equals()** method for our example. That is, we need to make our own **equals()** method for the **Movie** objects that overrides the one up in the **Object** class. To do this, we simply write the following method in the **Movie** class:

```
public boolean equals(Object x) {
      if (!(x instanceof Movie))
            return false;
      return title.equals(((Movie)x).title);
}
```

Notice that the method returns a **boolean** and takes a single parameter of type **Object**. This is a general parameter that allows us to compare the **Movie** with any type of object. Of course, if the parameter is not actually a **Movie** object, then the result should be **false**. This test is done using the **instanceof** keyword in JAVA that determines whether an object is an instance of a particular class.

Once we do that, we then need to decide what it means for two movies to be considered equal. For simplicity, we will assume that two **Movie** objects are equal if they have the same title. Notice that we simply ask the movies for their titles and then use the **equals()** method from the **String** class (already written to compare characters). One added point is that we need to *typecast* the parameter **x** to a **Movie** object.

Logically, the addition of this **equals()** method should solve the problem. However, it does not quite. As it turns out, in JAVA, **Sets** make use of a programming technique called ***hashing***. Hashing is used as a way of quickly comparing and sorting objects because it quickly identifies objects that cannot be equal to one another, without needing to go deep down inside the object to make comparisons. For example, if you had an apple and a pineapple, they are clearly not equal. You need not compare them closely because a simple quick glance tells you that they are not the same.

In real life, hashing is used by post offices when sorting mail at various levels. First, they look at the destination country and make two piles … domestic mail vs. international mail. That is a quick "hash" in that the postmen do not need to examine any further details at that time … such as street names and recipient names etc… Then they "hash" again later by using the *postal code* to determine "roughly" and "quickly" the area of a city that your mail needs to be delivered to. This allows them to make a pile of mail for all people living in the same area. At each level of "sorting" the mail (i.e., country, city, postal code, street), the postmen must make a quick decision as to which pile to place the mail item into. This quick decision is based on something called a ***hash function*** (or ***hash code***).

In JAVA, for **Sets** to work properly, we must also write a **hashCode()** method for our objects. These methods return an **int** which represents the "pile" that the object belongs to. Similar objects will have similar hash codes, and therefore end up in the same "pile". Here is a **hashCode()** method for our **Movie** object:

```
public int hashCode() {
    return title.hashCode();
}
```

It must be **public**, return an **int** and have no parameters. The code simply returns the hash code of the title string for the **Movie**. We do not wish to go into details here as to "how" to produce a proper hash code. Instead, let us simply use this "rule of thumb": the hash code for our objects should return a sum of all the hash codes of its attributes. If an attribute is a primitive, just convert it to an integer in some way and use that value in the **hashCode()** method's total value.

Now, the code in **SetTestProgram3** will work as desired ... removing all duplicates.

Example:

You will notice in the previous example, that the **HashSet** did not sort the items. Also, the items don't even appear in the order that they were added. Instead, the order seems somewhat random and arbitrary.

If you want the items in sorted order, you can use a **TreeSet** instead of a **HashSet**:

```
TreeSet<Movie>   displayList = new TreeSet<Movie>(inventory);
```

Of course, as we did with **PriorityQueues**, we will need to make sure that our **Movie** class implements the **Comparable<Movie>** interface and thus has a **compareTo()** method. Here is the completed **Movie** class that will work with both **HashSet** and **TreeSet**:

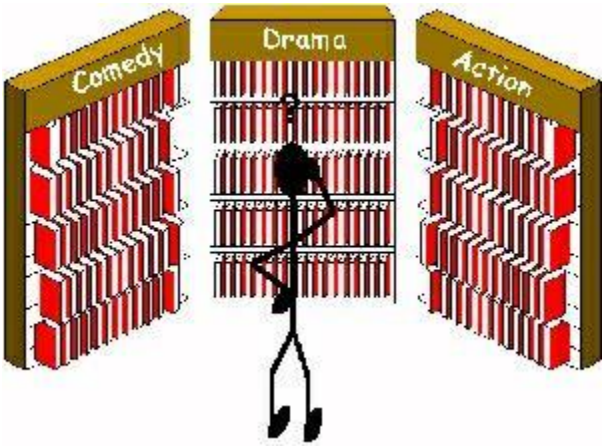
```
public class Movie implements Comparable<Movie> {
    private String  title;

    public Movie(String t) { title = t; }
    public String getTitle() { return title; }
    public String toString() { return "Movie: \"" + title + "\""; }

    public boolean equals(Object obj) {
        if (!(obj instanceof Movie)) return false;
        return title.equals(((Movie)obj).title);
    }

    public int hashCode() {
        return title.hashCode();
    }

    public int compareTo(Movie m) {
        return title.compareTo(m.title);
    }
}
```

Here is the output from our **SetTestProgram3** when using **TreeSet** instead of **HashSet**:

```
Here are the unique movies:
Movie: "Bolt"
Movie: "Hotel For Dogs"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "The Day the Earth Stood Still"

Here is the whole inventory:
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "The Day the Earth Stood Still"
Movie: "Marley & Me"
Movie: "Monsters Vs. Aliens"
Movie: "Hotel For Dogs"
Movie: "Monsters Vs. Aliens"
Movie: "Marley & Me"
Movie: "Bolt"
```

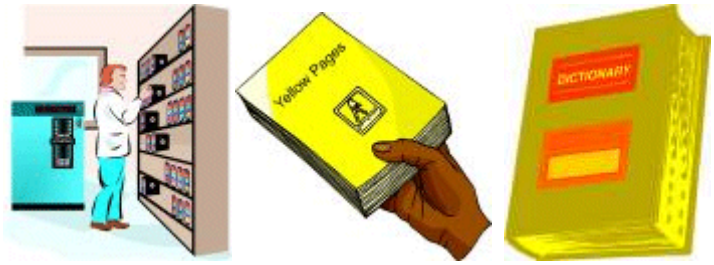
Notice the sorted order of the movies from the **TreeSet**.

8.7 The Dictionary/Map ADT



Sometimes we want to store data in an organized manner that allows us to quickly find what we are looking for. For example, consider a video store. Isn't it a nice idea to have the movies arranged by category so that you don't waste time looking over movies that are not of an interesting nature (such as musicals or perhaps children's movies) ?

You may also agree that it is easy to find a definition of a word in a dictionary because the definition is paired up with the word that it defines. Once we know the word that we want to look up, then we can find its definition. The **word** is therefore the unique ***key*** to finding the definition. We say that the **definition** of the word is the ***value*** associated with that key word. Likewise, a person's phone number is paired up with his/her name in a phonebook so that we can find numbers easily based the person's name as the key identifier. We say that the phone number is the ***value*** for the particular ***key*** person.



This idea of a key-value pairing (or mapping) is the basis of the **Dictionary** ADT:

*A **dictionary** is an abstract data type that stores a collection of unique keys and their associated values. Each **key** is associated with a single **value**.*

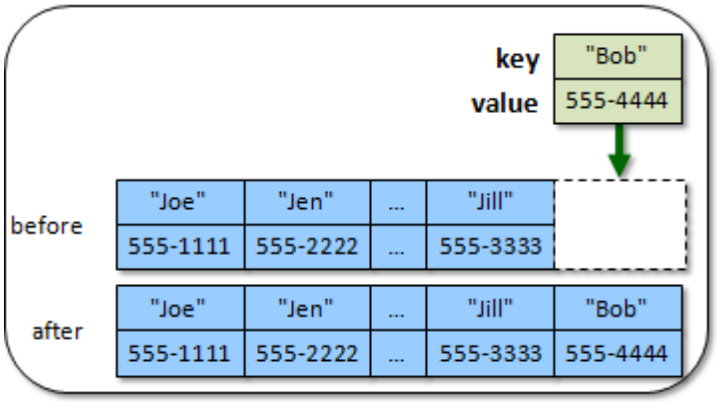
The key is always necessary in order to access a particular value in the dictionary. Like the **Collection** interface, the **Map** interface stores objects as well. So what is different ? Well, a **Map** stores things in a particular way such that the objects can be easily located later on. A ***Map*** really means a "Mapping" of one object to another. A **Map** does not keep items in any particular index order, as with lists. In a way, however, instead of having integer indices, a **Map** has arbitrary objects as indices. So just as a unique index into a list or array identifies the object at that location, the **key** in the **Map** identifies the object associated with it.

The basic methods for inserting, removing, accessing and modifying items from a **Map** are as follows:

put(Object k, Object v)

Place object **v** in the map with key **k**. If there is already a value at key **k**, it is replaced by **v**.

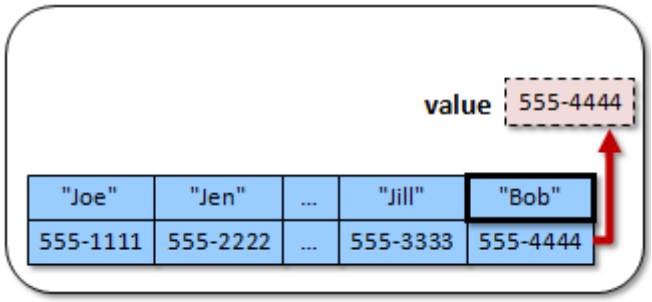
e.g., **aMap.put("Bob", "555-4444")** will do this →



get(Object k)

Return the value currently associated with key **k**. If **k** is not in the map yet, then **null** is returned.

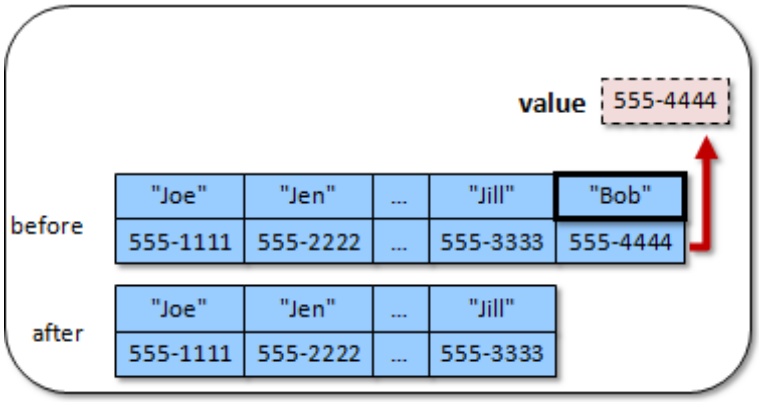
e.g., v = **aMap.get("Bob")** will return **v** →



remove(Object k)

Remove they key/value pair associated with key **k** from the Map. Usually, the value associated with the key is returned.

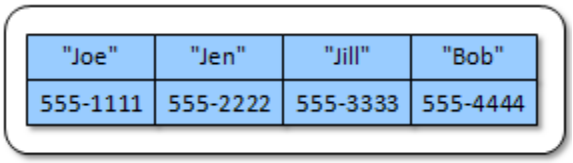
e.g., v = **aMap.remove("Bob")** will do this →



size()

Return the number of keys in the list.

e.g., n = **aMap.size()** will return **4** →



clear()

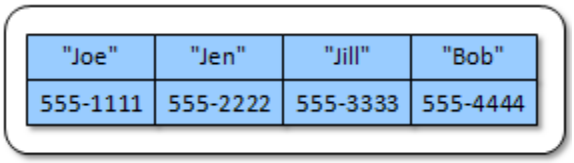
Remove all elements from the list.

There are additional methods often available for convenience sake. Here are some:

containsKey(Object k)

Return **true** if there is an entry in the map with key **k**, otherwise return **false**.

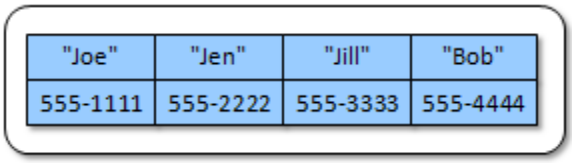
e.g., b = **aMap.containsKey("Bob")**
will return **true** →
e.g., b = **aMap.containsKey("Max")**
will return **false** →



containsValue(Object v)

Return **true** if there is an entry in the map with value **v**, otherwise return **false**.

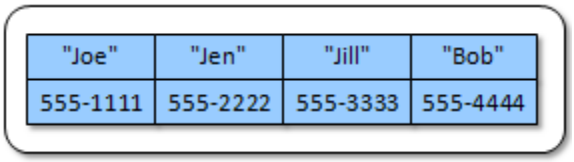
e.g., b = **aMap.containsValue("555-1111")**
will return **true** →
e.g., b = **aMap.containsValue("Jill")**
will return **false** →



keySet()

Return a **Set** containing all keys in the map.

e.g., s = **aMap.keySet()**
will return ["Joe", "Jen", "Jill", "Bob"] →



values()

Return a **Collection** of all values in the map.

e.g., c = **aMap.values()**
will return ["555-1111","555-2222",
"555-3333", "555-4444"] →

isEmpty()

Return **true** if the number of elements in the list is **0**, otherwise return **false**.

does the same as this:

```
return (aMap.size() == 0);
```

In JAVA, the *Map* ADT is called a **HashMap** and it is located in the **java.util** package, which must be imported in order to use this data type. To create a **HashMap**, we can simply call a constructor from the **HashMap** class. Here is an example of creating a **HashMap** and storing it in a variable so that we can use it:

```
HashMap  myMap;
myMap = new HashMap();
```

However, we usually indicate the type of the **key** and the **value** when we declare the map:

```
HashMap<String, Integer>   myMap;
myMap = new HashMap<String,Integer>();
```

There is also a **TreeMap** class that represents a sorted Map. That is, it maintains the keys in sorted order.

Example:

Consider a program that keeps track of some people and their favorite movie. We can associate a **Movie** object with each **Person** object and store them in a **HashMap** as follows:

```
import java.util.*;
public class MapTestProgram {
    public static void main(String args[]) {
        HashMap<Person, Movie> favMovies = new HashMap<Person, Movie>();
        Person pete, jack;

        favMovies.put(pete = new Person("Pete Zaria", 12),
                      new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Rita Book", 20),
                      new Movie("Marley & Me"));
        favMovies.put(new Person("Willie Maykit",65),
                      new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Patty O'Furniture", 41),
                      new Movie("Hotel For Dogs"));
        favMovies.put(new Person("Sue Permann", 73),
                      new Movie("Monsters Vs. Aliens"));
        favMovies.put(new Person("Sid Down", 19),
                      new Movie("Bolt"));
        favMovies.put(jack = new Person("Jack Pot", 4),
                      new Movie("Bolt"));

        System.out.println("There are: " + favMovies.size() + " favorite movies");
        System.out.println("Pete's favorite movie is: " + favMovies.get(pete));
        System.out.println("Jack's favorite movie is: " + favMovies.get(jack));
        System.out.println("The Map keys are:" + favMovies.keySet());
```

```
        System.out.println("The Map values are:" + favMovies.values());
        System.out.println("Removing Pete from the list ...");
        favMovies.remove(pete);
        System.out.println("Pete's favorite movie is: " + favMovies.get(pete));
        System.out.println("Is Pete in the Map ? " + favMovies.containsKey(pete));
        System.out.print("Is anyone's favorite Star Trek ? ");
        System.out.println(favMovies.containsValue(new Movie("Star Trek")));
        System.out.print("Is anyone's favorite Bolt ? ");
        System.out.println(favMovies.containsValue(new Movie("Bolt")));
    }
}
```

Here is the output:

```
There are: 7 favorite movies
Pete's favorite movie is: Movie: "Monsters Vs. Aliens"
Jack's favorite movie is: Movie: "Bolt"
The Map keys are:[4 year old Jack Pot, 65 year old Willie Maykit, 20 year old
Rita Book, 19 year old Sid Down, 73 year old Sue Permann, 41 year old Patty
O'Furniture, 12 year old Pete Zaria]
The Map values are:[Movie: "Bolt", Movie: "Monsters Vs. Aliens", Movie:
"Marley & Me", Movie: "Bolt", Movie: "Monsters Vs. Aliens", Movie: "Hotel For
Dogs", Movie: "Monsters Vs. Aliens"]
Removing Pete from the list ...
Pete's favorite movie is: null
Is Pete in the Map ? false
Is anyone's favorite Star Trek ? false
Is anyone's favorite Bolt ? true
```

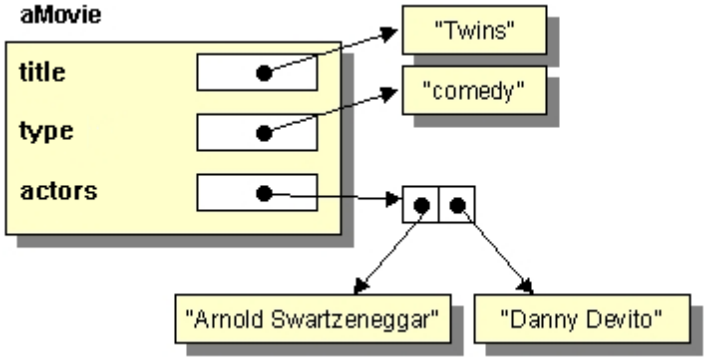
Example:

Consider an application which represents a movie store that maintains movies to be rented out. Assume that we have a collection of movies. When renting, we would like to be able to find movies quickly. For example, we may want to:

- ask for a movie by title and have it found right away
- search for movies in a certain category (e.g., new release, comedy, action)
- find movies containing a specific actor/actress (e.g., Jackie Chan, Peter Sellers etc...)

Obviously, we could simply store all moves in one big **ArrayList**. But how much time would we waste finding our movies ? Imagine a video store in which the movies are not sorted in any particular order ... just randomly placed on the shelves !! We would have to go through them one by one !!!

We will use **HashMaps** to store our movies efficiently so that we can quickly get access to the movies that we want. Let's start out with the representation of a **Movie** object. Each movie will maintain a **title**, list of **actors** and a **type** (category).



Obviously, in a real system, we would need to keep much more information such as ID, rental history, new releases vs. oldies, etc... Here is the diagram representing the **Movie** object:

Let us now define this **Movie** class.

```
import java.util.*;
public class Movie {
    private String             title, type;
    private ArrayList<String>  actors;

    public String getTitle() { return title; }
    public String getType() { return type; }
    public ArrayList<String> getActors() { return actors; }

    public Movie() {
        this("???", "???");
    }

    public Movie(String aTitle, String aType) {
        title = aTitle;
        type = aType;
        actors = new ArrayList<String>();
    }
    public String toString() { return("Movie: \"" + title + "\""); }
    public void addActor(String anActor) { actors.add(anActor); }
}
```

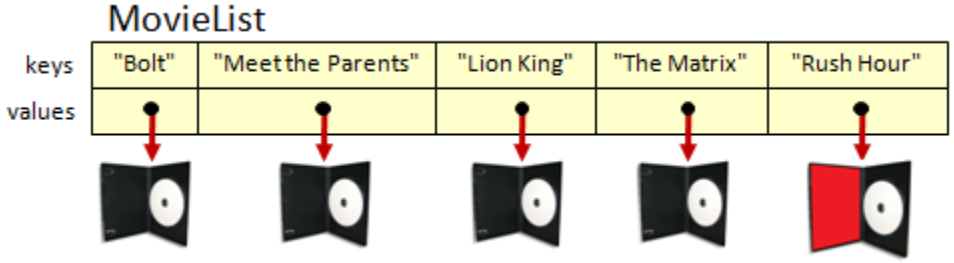
Now let's look at the **addActor()** method. It merely adds the given actor (just a name) to the actors arrayList. We can make some example methods to represent some movies. Add the following methods to the **Movie** class:

```
    public static Movie example1() {
        Movie aMovie = new Movie("The Matrix","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }
    public static Movie example2() {
        Movie aMovie = new Movie("Blazing Saddles","Comedy");
        aMovie.addActor("Cleavon Little");
        aMovie.addActor("Gene Wilder");
        return aMovie;
    }
    public static Movie example3() {
        Movie aMovie = new Movie("The Matrix Reloaded","SciFic");
        aMovie.addActor("Keanu Reeves");
        aMovie.addActor("Laurence Fishburne");
        aMovie.addActor("Carrie-Anne Moss");
        return aMovie;
    }
    public static Movie example4() {
        Movie aMovie = new Movie("The Adventure of Sherlock Holmes' Smarter Brother",
                                                                        "Comedy");
        aMovie.addActor("Gene Wilder");
        aMovie.addActor("Madeline Kahn");
        aMovie.addActor("Marty Feldman");
        aMovie.addActor("Dom DeLuise");
        return aMovie;
    }
```

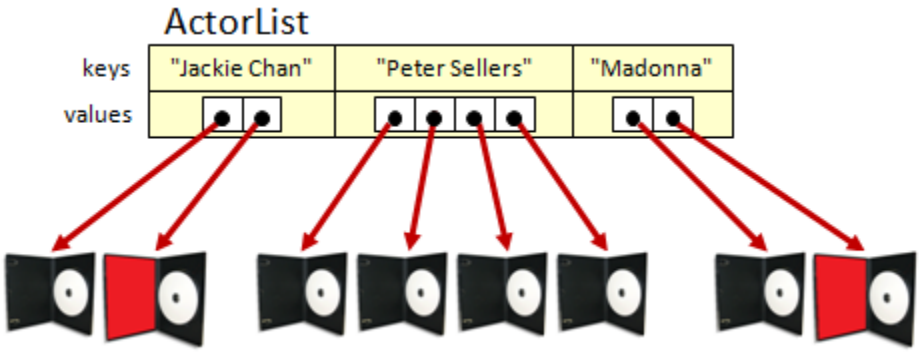
```
public static Movie example5() {
    Movie aMovie = new Movie("The Matrix Revolutions","SciFic");
    aMovie.addActor("Keanu Reeves");
    aMovie.addActor("Laurence Fishburne");
    aMovie.addActor("Carrie-Anne Moss");
    return aMovie;
}
public static Movie example6() {
    Movie aMovie = new Movie("Meet the Fockers","Comedy");
    aMovie.addActor("Robert De Niro");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Dustin Hoffman");
    return aMovie;
}
public static Movie example7() {
    Movie aMovie = new Movie("Runaway Jury","Drama");
    aMovie.addActor("John Cusack");
    aMovie.addActor("Gene Hackman");
    aMovie.addActor("Dustin Hoffman");
    return aMovie;
}
public static Movie example8() {
    Movie aMovie = new Movie("Meet the Parents","Comedy");
    aMovie.addActor("Robert De Niro");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Teri Polo");
    aMovie.addActor("Blythe Danner");
    return aMovie;
}
public static Movie example9() {
    Movie aMovie = new Movie("The Aviator","Drama");
    aMovie.addActor("Leonardo DiCaprio");
    aMovie.addActor("Cate Blanchett");
    return aMovie;
}
public static Movie example10() {
    Movie aMovie = new Movie("Envy","Comedy");
    aMovie.addActor("Ben Stiller");
    aMovie.addActor("Jack Black");
    aMovie.addActor("Rachel Weisz");
    aMovie.addActor("Amy Poehler");
    return aMovie;
}
```

Now we need to consider the making a **MovieStore** object. Recall, that we want to store movies efficiently using **HashMaps**.

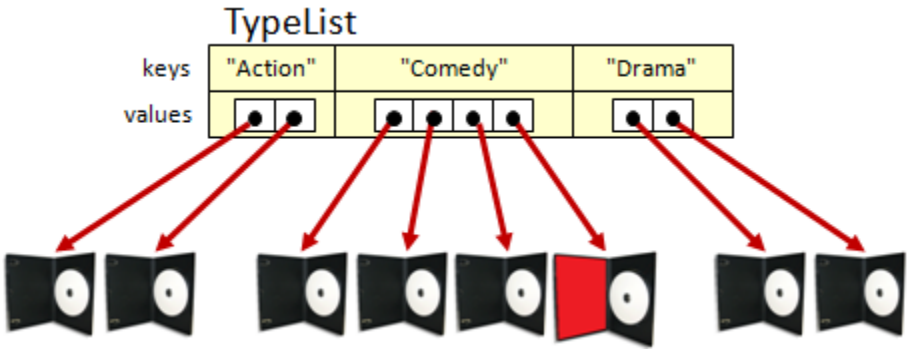
For the **MovieStore**, we will maintain three **HashMaps**. One will be the **movieList** where the keys are titles and the values are the movie objects with that title.



The second will be the **actorList** which will keep actor/actress names as keys and the values will be ArrayLists of all movies that the actor/actress stars in.



The last one will be the **typeList** in which the keys will be the "types" (or categories) of movies and the values will be ArrayLists of all movies belonging to that type.



Notice that one of the movies is "red" in the picture. Why ? This represents the same exact movie. So in fact, the reference to this movie is stored in 4 different places.

Isn't this wasteful ? Keep in mind that we are not duplicating all the movie's data ... we are only duplicating the pointer to the **Movie** object. So in fact, each time we duplicate a movie in our **HashMaps**, we are simply duplicating its reference (or pointer) which takes 4 bytes.

So, yes, we are taking slightly more space, but at the benefit of allowing quick access to the data. You will learn more about efficiency when you do your second-year course on data structures. The basic **MovieStore** definition is as follows:

```
import java.util.*;

public class MovieStore {
    private HashMap<String,Movie>              movieList;
    private HashMap<String,ArrayList<Movie>>   actorList;
    private HashMap<String,ArrayList<Movie>>   typeList;

    public MovieStore() {
        movieList = new HashMap<String,Movie>();
        actorList = new HashMap<String,ArrayList<Movie>>();
        typeList  = new HashMap<String,ArrayList<Movie>>();
    }
```

```
    public HashMap<String,Movie> getMovieList() { return movieList; }
    public HashMap<String,ArrayList<Movie>> getActorList() { return actorList; }
    public HashMap<String,ArrayList<Movie>> getTypeList() { return typeList; }

    public String toString() {
        return ("MovieStore with " + movieList.size() + " movies.");
    }
}
```

Why do we NOT need "set" methods for the **HashMaps** ? You should be able to reason on that. Now, how do we add a movie to the store ? Well ... how do the instance variables change ?

- movie must be added to movieList
- movie must be added to typeList. What if it is the first/last movie from this category ?
- movie must be added to actorList. What if it is the first/last movie for this actor ?

Here is the code:

```
//This method adds a movie to the movieStore.
public void addMovie(Movie aMovie) {
    //Add to the movieList
    movieList.put(aMovie.getTitle(), aMovie);

    //If there is no category matching this movie's type, make a new category
    if (!typeList.containsKey(aMovie.getType()))
        typeList.put(aMovie.getType(), new ArrayList<Movie>());

    //Add the movie to the proper category.
    typeList.get(aMovie.getType()).add(aMovie);

    //Now add all of the actors
    for (String anActor: aMovie.getActors()) {
        //If there is no actor yet matching this actor, make a new actor key
        if (!actorList.containsKey(anActor))
            actorList.put(anActor, new ArrayList<Movie>());

        //Add the movie for this actor
        actorList.get(anActor).add(aMovie);
    }
}
```

In fact, removing a movie is just as easy:

```
//This private method removes a movie from the movie store
private void removeMovie(Movie aMovie) {
    //Remove from the movieList
    movieList.remove(aMovie.getTitle());

    //Remove from the type list vector.  If last one, remove the type.
    typeList.get(aMovie.getType()).remove(aMovie);
    if (typeList.get(aMovie.getType()).isEmpty())
        typeList.remove(aMovie.getType());

    //Now Remove from the actors list.  If actor has no more, remove him.
```

```
    for(String anActor: aMovie.getActors()) {
        actorList.get(anActor).remove(aMovie);

        if (actorList.get(anActor).isEmpty())
            actorList.remove(anActor);
    }
}
```

However, what if we do not have a hold of the **Movie** object that we want to delete ? Perhaps we just know the title of the movie that needs to be removed. We can write a method which asks to remove a movie with a certain title. All it needs to do is grab a hold of the movie and then call the remove method that we just wrote.

```
//This method removes a movie (given its title) from the movie store
public void removeMovieWithTitle(String aTitle) {
    if (movieList.get(aTitle) == null)
        System.out.println("No movie with that title");
    else
        removeMovie(movieList.get(aTitle));
}
```

Well, perhaps the final thing we need to do is list the movies (or print them out). How do we do this ? What if we want them in some kind of order ? Perhaps any order, by actor/actress, or by type. Here's how to display them in the order that they were added to the **MovieStore**:

```
//This method lists all movie titles that are in the store
public void listMovies() {
    for (String s: movieList.keySet())
        System.out.println(s);
}
```

What about listing movies that star a certain actor/actress ? Well it just requires an additional search. Can you guess which **HashMap** is needed ?

```
//This method lists all movies that star the given actor
public void listMoviesWithActor(String anActor) {
    for (Movie m: actorList.get(anActor))
        System.out.println(m);
}
```

Lastly, let us list all of the movies that belong to a certain category (type). For example, someone may wish to have a list of all comedy movies in the store. It is actually very similar to the actor version.

```
//This method lists all movies that have the given type
public void listMoviesOfType(String aType) {
    for (Movie m: typeList.get(aType))
        System.out.println(m);
}
```

Ok, now we better test everything:

```
public class MovieStoreTester {
    public static void main(String args[]) {
        MovieStore  aStore = new MovieStore();
```

```
        aStore.addMovie(Movie.example1());
        aStore.addMovie(Movie.example2());
        aStore.addMovie(Movie.example3());
        aStore.addMovie(Movie.example4());
        aStore.addMovie(Movie.example5());
        aStore.addMovie(Movie.example6());
        aStore.addMovie(Movie.example7());
        aStore.addMovie(Movie.example8());
        aStore.addMovie(Movie.example9());
        aStore.addMovie(Movie.example10());

        System.out.println("Here are the movies in: " + aStore);
        aStore.listMovies();
        System.out.println();

        //Try some removing now
        System.out.println("Removing The Matrix");
        aStore.removeMovieWithTitle("The Matrix");
        System.out.println("Trying to remove Mark's Movie");
        aStore.removeMovieWithTitle("Mark's Movie");

        //Do some listing of movies
        System.out.println("\nHere are the Comedy movies in: " + aStore);
        aStore.listMoviesOfType("Comedy");
        System.out.println("\nHere are the Science Fiction movies in: " + aStore);
        aStore.listMoviesOfType("SciFic");
        System.out.println("\nHere are the movies with Ben Stiller:");
        aStore.listMoviesWithActor("Ben Stiller");
        System.out.println("\nHere are the movies with Keanu Reeves:");
        aStore.listMoviesWithActor("Keanu Reeves");
    }
}
```

Here is the output:

```
Here are the movies in: MovieStore with 10 movies.
The Matrix Revolutions
Runaway Jury
The Matrix
The Adventure of Sherlock Holmes' Smarter Brother
The Aviator
The Matrix Reloaded
Blazing Saddles
Meet the Parents
Envy
Meet the Fockers

Removing The Matrix
Trying to remove Mark's Movie
No movie with that title

Here are the Comedy movies in: MovieStore with 9 movies.
Movie: "Blazing Saddles"
Movie: "The Adventure of Sherlock Holmes' Smarter Brother"
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the Science Fiction movies in: MovieStore with 9 movies.
```

```
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"

Here are the movies with Ben Stiller:
Movie: "Meet the Fockers"
Movie: "Meet the Parents"
Movie: "Envy"

Here are the movies with Keanu Reeves:
Movie: "The Matrix Reloaded"
Movie: "The Matrix Revolutions"
```

8.8 *Collections* Class Tools

JAVA provides a nice tool-kit class called **Collections** that contains a bunch of useful methods that we can take advantage of. One of these is a **sort()** method which will sort an arbitrary collection.

Examine the following code to see how easy it is to sort our **ArrayList** of **Person** objects using this **sort()** method …

```
import java.util.*;

public class SortTestProgram {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete Zaria", 12));
        people.add(new Person("Rita Book", 20));
        people.add(new Person("Willie Maykit",65));
        people.add(new Person("Patty O'Furniture", 41));
        people.add(new Person("Sue Permann", 73));
        people.add(new Person("Sid Down", 19));
        people.add(new Person("Jack Pot", 4));

        Collections.sort(people);  // do the sorting

        for (Person p:  people)
            System.out.println(p);
    }
}
```

For the above code to work, we still need to have the **compareTo()** method written in the **Person** class. The output is as expected with all people sorted by their increasing age. Remember though, that we can set the **SortStrategy** to whatever we want. Hopefully you noticed how easy this **sort()** method is to use.

There is also a class called **Arrays** which has some useful methods for manipulating arrays. For example, if our code had arrays of **Person** objects instead of **ArrayLists**, here is what the code would look like to sort:

```
import java.util.*;

public class SortTestProgram2 {
    public static void main(String args[]) {
        Person[] people = {new Person("Pete Zaria", 12),
                new Person("Rita Book", 20),
                new Person("Willie Maykit",65),
                new Person("Patty O'Furniture", 41),
                new Person("Sue Permann", 73),
                new Person("Sid Down", 19),
                new Person("Jack Pot", 4)};

        Arrays.sort(people);    // do the sorting

        for (Person p:  people)
            System.out.println(p);
    }
}
```

There are similar sort methods for the primitive data types, so you can sort simple arrays of numbers such as this:

```
int[] nums = {23, 54, 76, 1, 29, 89, 45, 76};

Arrays.sort(nums);     // do the sorting
```

Interestingly, there are other useful methods in the **Collections** class such as **reverse()**, **shuffle()**, **max()** and **min()**. Can you guess what they do by looking at the output of the following program ?

```
import java.util.*;

public class SortTestProgram3 {
    public static void main(String args[]) {
        ArrayList<Person> people = new ArrayList<Person>();

        people.add(new Person("Pete Zaria", 12));
        people.add(new Person("Rita Book", 20));
        people.add(new Person("Willie Maykit",65));
        people.add(new Person("Patty O'Furniture", 41));
        people.add(new Person("Sue Permann", 73));
        people.add(new Person("Sid Down", 19));
        people.add(new Person("Jack Pot", 4));

        System.out.println("The list reversed:");
        Collections.reverse(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nThe list shuffled:");
        Collections.shuffle(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nThe list shuffled again:");
```

```
        Collections.shuffle(people);
        for(Person p:  people)
            System.out.println(p);

        System.out.println("\nOldest person: " + Collections.max(people));
        System.out.println("Youngest person:" + Collections.min(people));
    }
}
```

Here is the output … was it as you expected? …

```
The list reversed:
4 year old Jack Pot
19 year old Sid Down
73 year old Sue Permann
41 year old Patty O'Furniture
65 year old Willie Maykit
20 year old Rita Book
12 year old Pete Zaria

The list shuffled:
12 year old Pete Zaria
19 year old Sid Down
20 year old Rita Book
4 year old Jack Pot
65 year old Willie Maykit
41 year old Patty O'Furniture
73 year old Sue Permann

The list shuffled again:
65 year old Willie Maykit
20 year old Rita Book
4 year old Jack Pot
12 year old Pete Zaria
41 year old Patty O'Furniture
73 year old Sue Permann
19 year old Sid Down

Oldest person: 73 year old Sue Permann
Youngest person:4 year old Jack Pot
```

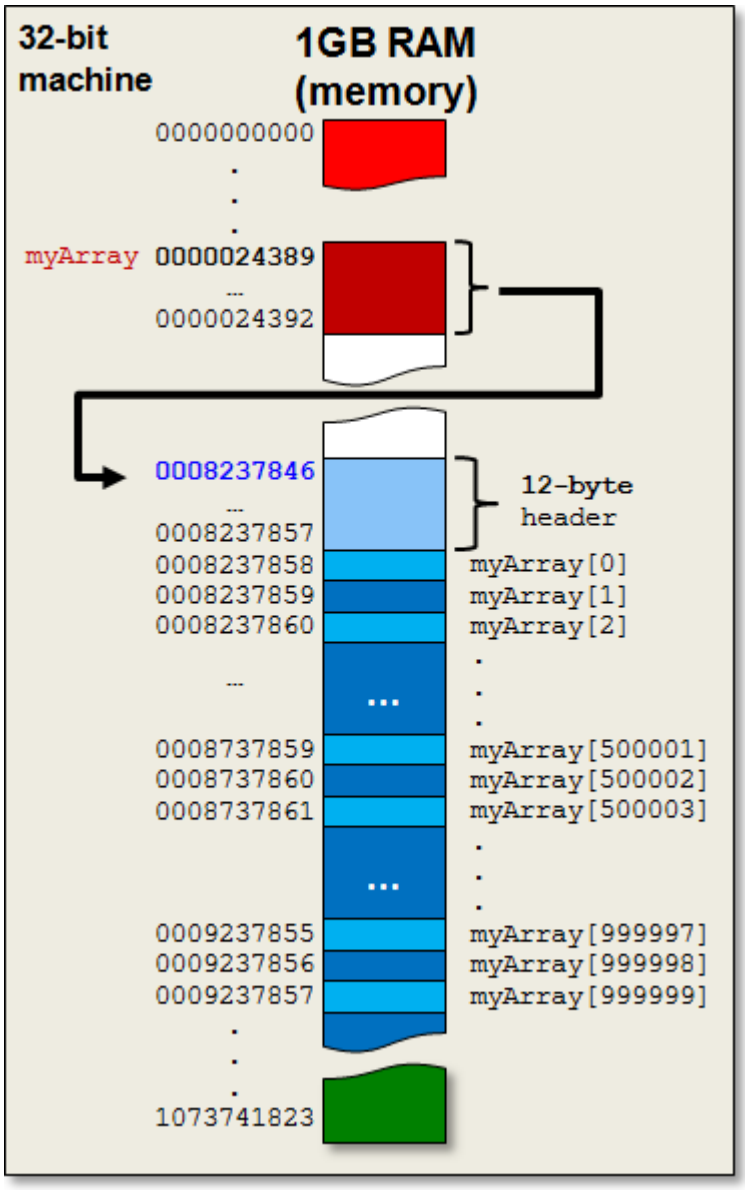
There are additional methods in the **Collections** class. Have a look at the API and see if you find anything useful.

8.9 Implementing an ADT (Doubly-Linked Lists)

Consider allocating a large array of bytes:

```
byte[]  myArray;

myArray = new byte[1000000];
```

An array is an object and the values of the array are kept in consecutive memory locations. Usually the array length is also kept along with the array, so we have shown this as an extra **4 bytes** in the object header ... making it a **12-byte** header (although the exact size depends on the java implementation).



Assume that this array is filled with some appropriate values. Consider what happens when we want to remove the item at position 500002 in the array.

Remember that we cannot simply remove something from an array but that we have 2 choices:

1. replace the item at index 500002 with some clearly identifiable value (e.g., -1).

2. remove the item at that position by copying all the items from index 500003 to 999999 back 1 position in the array and then reduce the size of the array by 1.

Solution (1) is quick to do a remove operation, but then we will leave "gaps" in the array so that when we process it later we will need to consider the fact that there may be a lot of invalid data (i.e., **nulls**) stored in the array at any time. In fact, after a while ... the array may be filled with mostly invalid data (i.e., **nulls**) !

Solution (2) would take a lot more time to remove an item because we would potentially need to move large portions of the array back one position in memory each time we do a remove operation. In addition, we can "logically" reduce the array size by one, but in reality, JAVA has already allocated the memory for the 1,000,000 elements ... so that will not change. In other words, we are essentially classifying the "end portion" of the array as **garbage data** as time goes on. We are not saving any space ... the garbage/wasted data is still taking up memory.

This problem gets worse as we consider adding items to the array beyond the 1,000,000 capacity. In that case, we would need to create a whole new bigger array and copy all the items from the "old" array into the "new" array ... then discard the old array. To do this, we would simply move the **myArray** variable pointer to point to the new array:

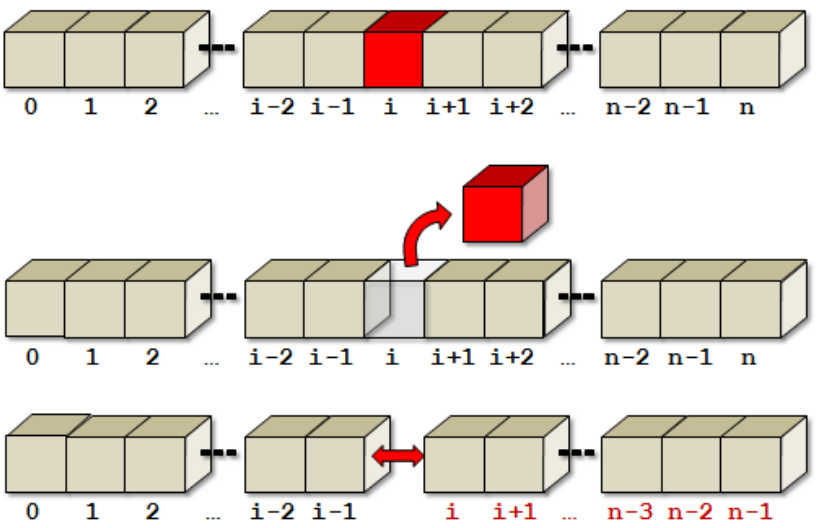
```
myArray = new byte[2000000];
```

This would create a whole new object which takes another 2,000,000 bytes (+12 for header). The Java VM would then realize that the data in memory from address 0008237846 to 0009237857 would no longer be needed and it would be scheduled for a future garbage collection operation. In languages such as C or C++, there is no garbage collector, so we would have to remember to free up that memory on our own.

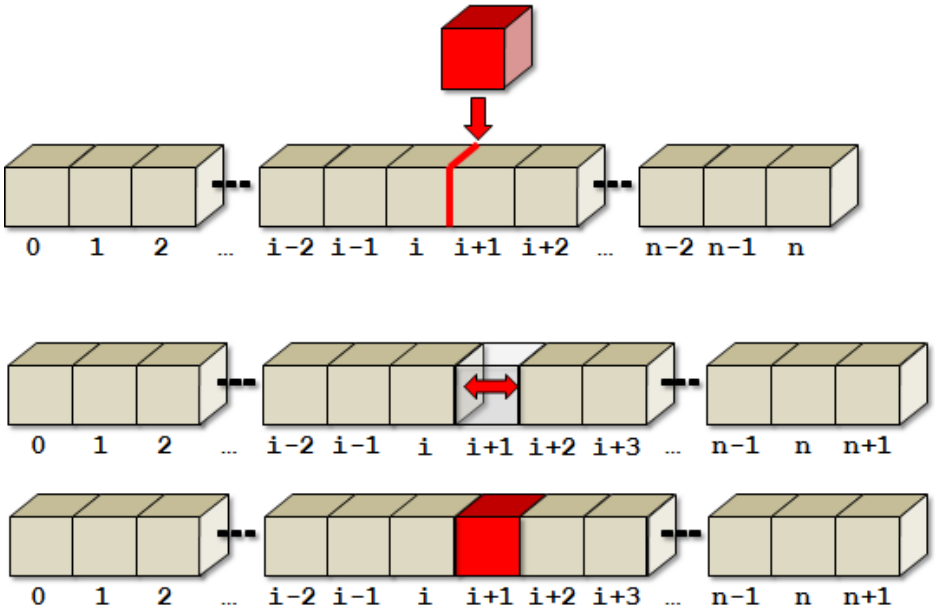
One huge danger of this "new-array-reallocation-and-copy-over" strategy is that if any other objects are pointing to the old array ... then it is not garbage collected and potentially we have two places in our code that at one time may have been pointing to the same array but are now pointing to different arrays !!

One solution to this problem is to store data in what is called a **doubly-linked list**. We would like to be able to do two things:

1. Cut out a single piece of data and stitch the remaining data back together:



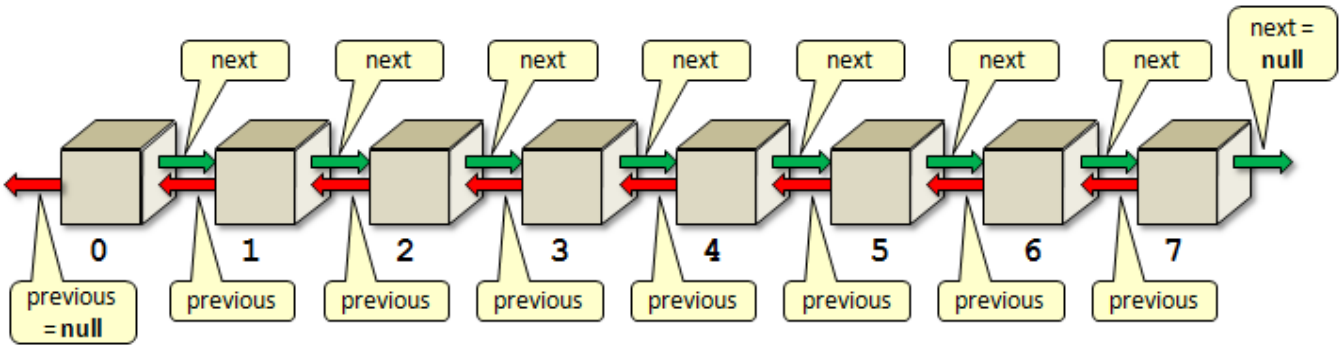
2. Cut open a spot in the data and insert a single piece of data inside:



To do this, we need to allow the data to be split and merged anywhere within the list of data. We can do this by allowing each piece of data to be its own object. As long as each of these objects knows the object before it in the list as well as the object after it in the list, then we can make this happen. Consider a single item in the list represented as follows:

```
public class Item {
    byte   data;
    Item   previous;
    Item   next;

    public Item(int d) {
        data = (byte)d;
        previous = null;
        next = null;
    }
}
```

Notice that this **Item** class represents a *recursive data structure* definition since the item before (i.e., *previous*) this item is an **Item** object and the item after (i.e., *next*) it is also an **Item** object. That means, the items each keep a pointer to the object before it in the list and the object after it in the list. So we can re-draw our **n-item** list now as follows:



Consider a simple array created as follows:

```
byte[]      myList = new byte[8];
myList[0] = 23; myList[1] = 65;
myList[2] = 87; myList[3] = 45;
myList[4] = 56; myList[5] = 34;
myList[6] = 95; myList[7] = 71;
```

Here is how we would create the linked-list version for this list of data:

```
Item  myList  = new Item(23);
Item  myList1 = new Item(65);
Item  myList2 = new Item(87);
Item  myList3 = new Item(45);
Item  myList4 = new Item(56);
Item  myList5 = new Item(34);
Item  myList6 = new Item(95);
Item  myList7 = new Item(71);

myList.previous = null;
myList.next = myList1;
myList1.previous = myList;
myList1.next = myList2;
myList2.previous = myList1;
myList2.next = myList3;
myList3.previous = myList2;
myList3.next = myList4;
myList4.previous = myList3;
myList4.next = myList5;
myList5.previous = myList4;
myList5.next = myList6;
myList6.previous = myList5;
myList6.next = myList7;
myList7.previous = myList6;
myList7.next = null;
```

This code is a bit ugly because it uses many variable names. However, typically we would create operations for adding and removing Items. Also, we usually want to keep track of the **first** and **last** items in the linked list ... which are known as the ***head*** and the ***tail***. So, often we create another class to keep track of this information as follows:

```
public class LinkedList {
    Item   head;
    Item   tail;

    public LinkedList() {
        head = null;
        tail = null;
    }
}
```

Then we would make operations in this class.

One useful operation would be to add an item to the end (i.e., tail) of the list.

Here is the code that will do this:

```
public void add(Item x) {
	if (tail == null) {
		tail = x;
		head = x;
	}
	else {
		tail.next = x;
		x.previous = tail;
		tail = x;
		x.next = null;  // In case it was an Item taken from another list
	}
}
```

Notice that we had to handle the case where we called the method the very first time. In that case, the **head** and the **tail** would both be **null**. So, when adding in that case, the item being added becomes the sole item in the list ... making it both the **head** and the **tail** at the same time. From then on, all additions occur at the **tail** end of the list.

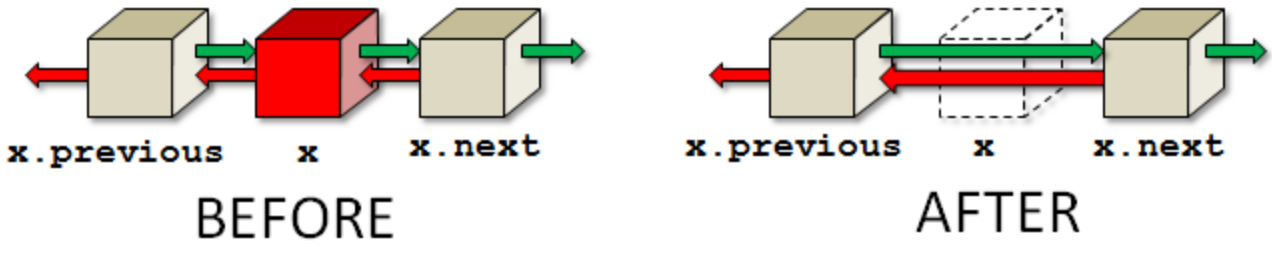
Once we have this method available, the code to construct the list becomes simplified:

```
LinkedList myList = new LinkedList();
myList.add(new Item(23));
myList.add(new Item(65));
myList.add(new Item(87));
myList.add(new Item(45));
myList.add(new Item(56));
myList.add(new Item(34));
myList.add(new Item(95));
myList.add(new Item(71));
```

Is this better than an array ? It seems like a lot of overhead! Well ... it may indeed take up more space ... but the size of the list is unlimited (except for running out of computer memory). That is ... we never have to worry about going past an array bounds. Also, we never have to worry about re-allocating a new larger array and copying elements over into it.

What about the removal of an item from the list ?

It too just involves moving a couple of pointers around.



Here is the code to remove an item ... assuming that **x** is in the list:

```
    public void remove(Item x) {
        if (x == head) {
            if (x == tail) {
                head = tail = null;
            }
            else {
                head = x.next;
                head.previous = null;
            }
        }
        else {
            if (x == tail) {
                tail = x.previous;
                tail.next = null;
            }
            else {
                x.previous.next = x.next;
                x.next.previous = x.previous;
            }
        }
    }
```

The code looks a little long because we need to handle the special cases in which the removed item is the **head** of the list or the **tail** of the list. However, you will notice that the code for removal simply involves the changing of two pointers. There is no need to copy items back in the array, nor is there any concern about garbage data lying around. The code is quite simple and elegant.

How could we write a **toString()** method for this list that shows the contents ? Assume that we want the list to look like this:

[H:23]<==>[65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

or like this when 1 item is in it: [H:23:T]
or like this when empty: [EMPTY]

To iterate through the items, we would need to start with the **head** of the list and keep traversing successive .**next** pointers until we reached the **tail**.

```
public String toString() {
    if (head == null)
        return "[EMPTY]";

    String s = "[H:";
    Item currentItem = head;
    while (currentItem != null) {
        s += currentItem.data;
        if (currentItem != tail)
            s += "]<==>[";
        currentItem = currentItem.next;
    }
    return s + ":T]";
}
```

How would we write a method to add up all of the byte data in the list? It is quite similar:

```
   public int totalData() {
        if (head == null)
             return 0;

        int  total = 0;
        Item currentItem = head;
        while (currentItem != null) {
             total += currentItem.data;
             currentItem = currentItem.next;
        }
        return total;
   }
```

Do you understand why a **FOR** loop was not used ?

Here is a test program:

```
public class LinkedListTestProgram {
    public static void main(String args[]) {
        Item   head, tail, internal;

        LinkedList myList = new LinkedList();
        myList.add(head = new Item(23));
        myList.add(new Item(65));
        myList.add(new Item(87));
        myList.add(internal = new Item(45));
        myList.add(new Item(56));
        myList.add(new Item(34));
        myList.add(new Item(95));
        myList.add(tail = new Item(71));

        System.out.println("Here is the list: ");
        System.out.println(myList);

        System.out.println("\nThe total of the data is: ");
        System.out.println(myList.totalData());

        System.out.println("\nRemoving the head .. here is the list now: ");
        myList.remove(head);
        System.out.println(myList);

        System.out.println("\nRemoving the tail .. here is the list now: ");
        myList.remove(tail);
        System.out.println(myList);

        System.out.println("\nRemoving internal item 45, here is the list now: ");
        myList.remove(internal);
        System.out.println(myList);
    }
}
```

Here is the output:

```
Here is the list:
[H:23]<==>[65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

The total of the data is:
476

Removing the head .. here is the list now:
[H:65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95]<==>[71:T]

Removing the tail .. here is the list now:
[H:65]<==>[87]<==>[45]<==>[56]<==>[34]<==>[95:T]

Removing internal item 45, here is the list now:
[H:65]<==>[87]<==>[56]<==>[34]<==>[95:T]
```

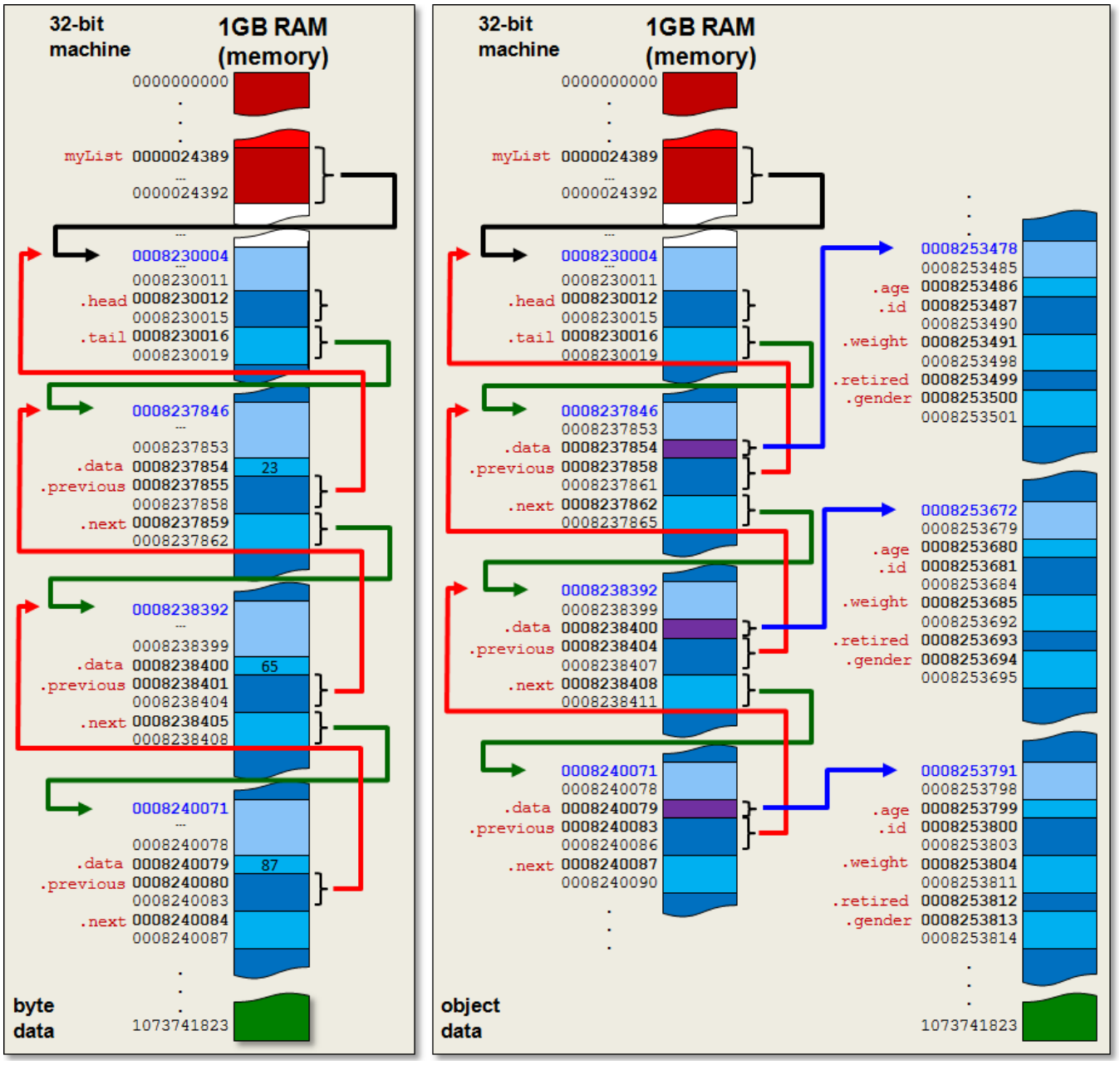
Can you write an **insert(x, i)** method that will insert item **x** after position **i** in the list ? Try it. You will need to start at the **head** of the list and count past **i** items before you start changing pointers. Can you do a **remove(i)** method that will remove the **i**'th item from the list ?

It is important to understand how to manipulate pointers like this because some languages (e.g., C and C++) require a lot of memory allocation and pointer manipulation. The more practice you get ... the better!!

You should realize that although our list contained simple data in the form of a single **byte**, you can simply change the type of the **data** to any data type. In this way, the list can store any kind of data that you want. Here is a general definition for a list Item that can store any object:

```
public class Item {
    Object data;
    Item   previous;
    Item   next;

    public Item(Object obj) {
        data = obj;
        previous = null;
        next = null;
    }
}
```

Notice what the memory allocation would look like for a simple 3-item list when simple **byte** data is used (left side diagram) and when **Person** object data is used (right side diagram):


32-bit machine
1GB RAM (memory)
0000000000
myList 0000024389
0000024392
0008230004
0008230011
.head 0008230012
0008230015
.tail 0008230016
0008230019
0008237846
0008237853
.data 0008237854
.previous 0008237855
0008237858
.next 0008237859
0008237862
23
0008238392
0008238399
.data 0008238400
.previous 0008238401
0008238404
.next 0008238405
0008238408
65
0008240071
0008240078
.data 0008240079
.previous 0008240080
0008240083
.next 0008240084
0008240087
87
1073741823
byte data
32-bit machine
1GB RAM (memory)
0000000000
myList 0000024389
0000024392
0008230004
0008230011
.head 0008230012
0008230015
.tail 0008230016
0008230019
0008237846
0008237853
.data 0008237854
.previous 0008237858
0008237861
.next 0008237862
0008237865
0008238392
0008238399
.data 0008238400
.previous 0008238404
0008238407
.next 0008238408
0008238411
0008240071
0008240078
.data 0008240079
.previous 0008240083
0008240086
.next 0008240087
0008240090
0008253478
0008253485
.age 0008253486
.id 0008253487
0008253490
.weight 0008253491
0008253498
.retired 0008253499
.gender 0008253500
0008253501
0008253672
0008253679
.age 0008253680
.id 0008253681
0008253684
.weight 0008253685
0008253692
.retired 0008253693
.gender 0008253694
0008253695
0008253791
0008253798
.age 0008253799
.id 0008253800
0008253803
.weight 0008253804
0008253811
.retired 0008253812
.gender 0008253813
0008253814
1073741823
object data