Accessing Your Stored Java Objects with the Iterator Design Pattern

by Barry A. Burd and Michael P. Redlich

Introduction

This article, the eighth in a series about design patterns, introduces the Iterator design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of the book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known as the *Gang of Four* (GoF). The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Iterator design pattern fits into the behavioral category. According to the GoF book, the Iterator design pattern "provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation."

Motivation

Imagine that you're in charge of a sports application. You maintain the teams from one sport (such as baseball) in an **`ArrayList`**, and maintain teams from another sport (maybe football) in an ordinary Java array. With your access to both teams' data structures you to manage all data in the application. Here's some rudimentary code:

```
import java.text.DecimalFormat;

public class Sports {
      private String team;
      private int win;
      private int loss;
      private int tie;
      private double pct;

      public Sports() {
            setTeam("Team");
            setWin(0);
            setLoss(0);
            setTie(0);
            setPct(0,0,0);
            }

      public Sports(String team,int win,int loss,int tie) {
            setTeam(team);
            setWin(win);
            setLoss(loss);
            setTie(tie);
            setPct(win,loss,tie);
            }

      public String getTeam() {
            return team;
            }
```

```
	public void setTeam(String team) {
		this.team = team;
		}

	public int getWin() {
		return win;
		}

	public void setWin(int win) {
		this.win = win;
		}

	public int getLoss() {
		return loss;
		}

	public void setLoss(int loss) {
		this.loss = loss;
		}

	public int getTie() {
		return tie;
		}

	public void setTie(int tie) {
		this.tie = tie;
		}

	public double getPct() {
		return pct;
		}

	public void setPct(int win,int loss,int tie) {
		if(win == 0 && loss == 0) {
			this.pct = 0.0;
			}
		else {
			this.pct = (double)win / ((double)win + (double)loss + (double)tie);
			}
		}

	public String toString() {
		DecimalFormat df = new DecimalFormat("0.000");
		return String.format("%-25s",getTeam())
			+ String.format("%5d",getWin())
			+ String.format("%5d",getLoss())
			+ String.format("%5d",getTie()) + "\t"
			+ df.format(getPct());
		}
	}
```

Listing 1: The Sports base class

```
import java.text.DecimalFormat;

public class Baseball extends Sports {
	public Baseball() {
		super();
		}

	public Baseball(String team,int win,int loss) {
		super(team,win,loss,0);
```

```
            }

      public String toString() {
            DecimalFormat df = new DecimalFormat("0.000");
            return String.format("%-25s",getTeam())
                  + String.format("%5d",getWin())
                  + String.format("%5d",getLoss()) + "\t\t"
                  + df.format(getPct());
            }
      }
```

Listing 2: The Baseball class

```
public class Football extends Sports {
      public Football() {
            super();
            }

      public Football(String team,int win,int loss,int tie) {
            super(team,win,loss,tie);
            }
      }
```

Listing 3: The Football class

Listing 1 shows a base class, **`Sports`**, which handles the crux of sports-related activities (i.e., getters and setters) for team information such as team name, number of wins, losses, ties, and winning percentage (which is automatically calculated). The derived classes **`Baseball`** and **`Football`**, shown in Listings 2 and 3, use **`super()`** for object creation.

Notice the difference between the **`Baseball`** and **`Football`** classes. The **`Baseball`** class defines its own **`toString()`** method. Sports fans know that there are no ties in baseball. (Do you remember the classic 24-inning games of the past?) So the **`Baseball`** class's **`toString()`** method doesn't call **`getTie()`**. Also, the **`Baseball`** class's second constructor has no **`tie`** parameter. The constructor sneaks the value **`0`** for **`tie`** in a **`super()`** call.

In Listing 4, the **`BaseballTeams`** class collects **`Baseball`** objects into a group of some kind. In Listing 5, the **`FootballTeams`** class does the same for **`Football`** objects.

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

public class BaseballTeams {
      private List<Sports> baseballList;
      private static final String baseballFile = "mlb2006.csv";

      public BaseballTeams() {
            buildTeamList();
            }

      public void buildTeamList() {
            baseballList = new ArrayList<Sports>();
            try {
                  BufferedReader file = new BufferedReader(new FileReader(baseballFile));
                  String line = null;
```

```
                while((line = file.readLine()) != null) {
                      String team = null;
                      int win = 0;
                      int loss = 0;
                      StringTokenizer tokenizer = new StringTokenizer(line,",\n");
                      while(tokenizer.hasMoreTokens()) {
                            team = tokenizer.nextToken();
                            win = new Integer(tokenizer.nextToken());
                            loss = new Integer(tokenizer.nextToken());
                            }
                      Sports baseball = new Baseball(team,win,loss);
                      baseballList.add(baseball);
                      }
                file.close();
                }
          catch(FileNotFoundException exception) {
                exception.printStackTrace();
                }
          catch(IOException exception) {
                exception.printStackTrace();
                }
          }

     public List<Sports> getTeamList() {
          return baseballList;
          }
     }
```

Listing 4: The BaseballTeams class

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class FootballTeams {
     private static final int MAX = 32;
     private Sports[] footballList;
     private static final String footballFile = "nfl2006.csv";

     public FootballTeams() {
          buildTeamList();
          }

     public void buildTeamList() {
          footballList = new Sports[MAX];
          try {
                BufferedReader file = new BufferedReader(new FileReader(footballFile));
                String line = null;
                int current = 0;
                while((line = file.readLine()) != null) {
                      String team = null;
                      int win = 0;
                      int loss = 0;
                      int tie = 0;
                      StringTokenizer tokenizer = new StringTokenizer(line,",\n");
                      while(tokenizer.hasMoreTokens()) {
                            team = tokenizer.nextToken();
                            win = new Integer(tokenizer.nextToken());
                            loss = new Integer(tokenizer.nextToken());
                            tie = new Integer(tokenizer.nextToken());
                            }
```

```
                        Sports football = new Football(team,win,loss,tie);
                        if(current < MAX) {
                              footballList[current] = football;
                              ++current;
                              }
                        else {
                              System.err.println("--> ERROR: Array is full!");
                              }
                        }
                  file.close();
                  }
            catch(FileNotFoundException exception) {
                  exception.printStackTrace();
                  }
            catch(IOException exception) {
                  exception.printStackTrace();
                  }
            }

      public Sports[] getTeamList() {
            return footballList;
            }
      }
```

Listing 5: The FootballTeams class

In the real world, you'd store team information in a database. But this simple example parses small comma-separated value (CSV) files -- the files **`mlb2006.csv`** and **`nfl2006.csv`** named in Listings 4 and 5.[1] Building the teams for each sport is relatively straightforward: You read the CSV files into a Java **`BufferedReader`**, parse the data using a **`StringTokenizer`**, and then store the data in two data structures (an **`ArrayList`** for baseball and a Java array for football).

Listing 6 shows a client application.

```
import java.util.List;

public class SportsApp {
      public static void main(String[] args) {
            // output Major League Baseball teams
            BaseballTeams baseball = new BaseballTeams();
            List<Sports> baseballList = baseball.getTeamList();
            System.out.println("### Major League Baseball 2006 ###\n");
            for(int i = 0;i < baseballList.size();++i) {
                  System.out.println(baseballList.get(i));
                  }

            System.out.println();

            // output National Football League teams
            FootballTeams football = new FootballTeams();
            Sports[] footballList = football.getTeamList();
            System.out.println("### National Football League 2006 ###\n");
            for(int i = 0;i < footballList.length;++i) {
                  System.out.println(footballList[i]);
                  }
            }
      }
```

[1] You might be wondering why we aren't using final 2007 standings. After all, this is 2008! Unfortunately, Mike is still smarting from the Mets September 2007 collapse and it would pain him to see the Mets listed in second place.

Listing 6: The SportsApp class – a client application

Running the application produces the following output as shown in Figure 1.

```
### Major League Baseball 2006 ###

New York Yankees             97   65       0.599
Toronto Blue Jays            87   75       0.537
Boston Red Sox               86   76       0.531
Baltimore Orioles            70   92       0.432
Tampa Bay Devil Rays         61  101       0.377
Minnesota Twins              96   66       0.593
Detroit Tigers               95   67       0.586
Chicago White Sox            90   72       0.556
Cleveland Indians            78   84       0.481
Kansas City Royals           62  100       0.383
Oakland A's                  93   69       0.574
Los Angeles Angels           89   73       0.549
Texas Rangers                80   82       0.494
Seattle Mariners             78   84       0.481
New York Mets                97   65       0.599
Philadelphia Phillies        85   77       0.525
Atlanta Braves               79   83       0.488
Florida Marlins              78   84       0.481
Washington Nationals         71   91       0.438
St. Louis Cardinals          83   78       0.516
Houston Astros               82   80       0.506
Cincinnati Reds              80   82       0.494
Milwaukee Brewers            75   87       0.463
Pittsburgh Pirates           67   95       0.414
Chicago Cubs                 66   96       0.407
San Diego Padres             88   74       0.543
Los Angeles Dodgers          88   74       0.543
San Francisco Giants         76   85       0.472
Arizona Diamondbacks         76   86       0.469
Colorado Rockies             76   86       0.469

### National Football League 2006 ###

New England Patriots         12    4    0  0.750
New York Jets                10    6    0  0.625
Buffalo Bills                 7    9    0  0.438
Miami Dolphins                6   10    0  0.375
Baltimore Ravens             13    3    0  0.812
Cincinnati Bengals            8    8    0  0.500
Pittsburgh Steelers           8    8    0  0.500
Cleveland Browns              4   12    0  0.250
Indianapolis Colts           12    4    0  0.750
Tennessee Titans              8    8    0  0.500
Jacksonville Jaguars          8    8    0  0.500
Houston Texans                6   10    0  0.375
San Diego Chargers           14    2    0  0.875
Kansas City Chiefs            9    7    0  0.562
Denver Broncos                9    7    0  0.562
Oakland Raiders               2   14    0  0.125
Philadelphia Eagles          10    6    0  0.625
Dallas Cowboys                9    7    0  0.562
New York Giants               8    8    0  0.500
Washington Redskins           5   11    0  0.312
Chicago Bears                13    3    0  0.812
Green Bay Packers             8    8    0  0.500
Minnesota Vikings             6   10    0  0.375
Detroit Lions                 3   13    0  0.188
```

```
New Orleans Saints          10    6    0  0.625
Carolina Panthers            8    8    0  0.500
Atlanta Falcons              7    9    0  0.438
Tampa Bay Buccaneers         4   12    0  0.250
Seattle Seahawks             9    7    0  0.562
St. Louis Rams               8    8    0  0.500
San Francisco 49ers          7    9    0  0.438
Arizona Cardinals            5   11    0  0.312
```

Figure 1: The output of the Sports application

Adding an additional sport (basketball, hockey, pie-eating, or whatever) isn't difficult. But Listings 1 through 6 have a very serious deficiency. The client code knows way too much about the implementation (the underlying data structures) of the Sports application. In particular, the client code (Listing 6) uses an **`ArrayList`** for baseball

```
List<Sports> baseballList = baseball.getTeamList();
```

And for football, the client code uses a Java array.

```
Sports[] footballList = football.getTeamList();
```

The difference between an **`ArrayList`** and an array forces the client code's **`for`**-loops to differ slightly:

```
for(int i = 0;i < baseballList.size();++i) { ...

for(int i = 0;i < footballList.length;++i) { ...
```

These code differences are clumsy and wasteful. As you may expect, the Iterator design pattern comes to the rescue.

UML Diagram

Figure 2 shows a UML diagram for the Iterator design pattern.

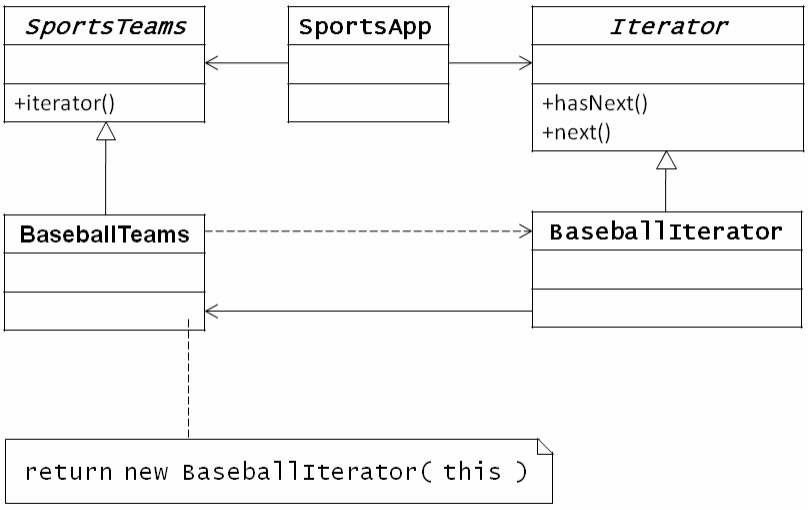

Figure 2: The UML diagram for the Iterator design pattern

The diagram contains five classes:

- The **Iterator** interface declares methods **hasNext()** and **next()**.
- The concrete **BaseballIterator** class implements **hasNext()** and **next()** for the data structure that stores baseball teams. (Likewise, the concrete **FootballIterator** class implements **hasNext()** and **next()** for the data structure that stores football teams.)
- The **SportsTeams** interface declares the **iterator()** method. This method returns an **Iterator** object -- an object with the appropriate **hasNext()** and **next()** methods. In the sports application, an "appropriate" method is one that works with either a **List<Sports>** object or a **Sports[]** array depending on the kind of team (baseball or football).

In the official GoF book's lingo, an interface such as **SportsTeams** is called an *aggregate*. An aggregate houses a group of objects, perhaps a Java collection of some kind (an **ArrayList** of baseball teams, for example). The iterator's job is to traverse the items that the aggregate stores. Now, back to the UML diagram in Figure 2:

- The concrete **BaseballTeams** class implements the **iterator()** method for the data structure that stores baseball teams. (Likewise, the concrete **FootballTeams** class implements **iterator()** for the data structure that stores football teams.)
- The **SportsApp** class, the client application, uses aggregates and their iterators.

Implementing the Iterator Design Pattern

Listing 7 contains an **Iterator** interface.

```
public interface Iterator<T> {
      public boolean hasNext();
      public T next();
      }
```

Listing 7: The Iterator interface

In Listing 7, the **next()** method, returns the next element in the data structure. The method's return type is **T**, a generic data type representing a **BaseballTeam**, a **FootballTeam**, or whatever. The **hasNext()** method returns either **true** (meaning "yes, the client can fetch another item from the aggregate") or **false** (meaning "no, the aggregate has no items worth fetching, because the client has already fetched all the aggregate's items").

Listing 8 defines an aggregate interface. This particular aggregate contains teams.

```
public interface SportsTeams {
      public void buildTeamList();
      public Iterator<Sports> iterator();
      }
```

Listing 8: The SportsTeams interface

In Listing 8, the new **iterator()** method returns a concrete instance of **Iterator<Sports>**. The return type isn't **Iterator<Baseball>** or **Iterator<Football>** because the code uses an interface and a base class.

The next order of business is to create concrete classes from the **Iterator** interface. The classes are in Listings 9 and 10.

```
import java.util.List;

public class BaseballIterator implements Iterator<Sports> {
      private int current = 0;
      private List<Sports> baseballList;

      public BaseballIterator(List<Sports> baseballList) {
            this.baseballList = baseballList;
            }

      public boolean hasNext() {
            if(current >= baseballList.size() || baseballList.get(current) == null) {
                  return false;
                  }
            else {
                  return true;
                  }
            }

      public Sports next() {
            Sports baseball = baseballList.get(current);
            ++current;
            return baseball;
            }
      }
```

Listing 9: The BaseballIterator class

```
public class FootballIterator implements Iterator<Sports> {
      private int current = 0;
      private Sports[] footballList;

      public FootballIterator(Sports[] footballList) {
            this.footballList = footballList;
            }

      public boolean hasNext() {
            if(current >= footballList.length || footballList[current] == null) {
                  return false;
                  }
            else {
                  return true;
                  }
            }

      public Sports next() {
            Sports football = footballList[current];
            ++current;
            return football;
            }
      }
```

Listing 10: The FootballIterator class

In Listing 9 the **`BaseballIterator`** class has its own **`baseballList`** -- an instance of **`java.util.List<Sports>`**. In Listing 10, the **`FootballIterator`** class has its own **`footballList`** -- an instance of **`Sports[]`**. The use of instances is called *composition*. (The alternative to composition is *subclassing*, in which **`BaseballIterator`** and **`FootballIterator`** are subclasses of **`Sports`** collections. For the Iterator pattern, composition is much better than subclassing.) The **`BaseballIterator`** and **`FootballIterator`** classes use composition instead of subclassing.

Listings 4 and 5 don't use the Iterator pattern. These listings contain two different **`getTeamList()`** methods:

```
public List<Sports> getTeamList() // baseball
public Sports[] getTeamList() // football
```

Using the Iterator pattern, Listings 11 and 12 replace these **`getTeamList()`** methods with one **`iterator()`** method:

```
public Iterator<Sports> iterator() // both baseball and football
```

Listings 11 and 12 have the improved code using the uniform **`iterator()`** method:

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import java.util.StringTokenizer;

public class BaseballTeams implements SportsTeams {
      private List<Sports> baseballList;
      private static final String baseballFile = "mlb2006.csv";

      public BaseballTeams() {
            buildTeamList();
            }

      public void buildTeamList() {
            baseballList = new ArrayList<Sports>();
            try {
                  BufferedReader file = new BufferedReader(new FileReader(baseballFile));
                  String line = null;
                  while((line = file.readLine()) != null) {
                        String team = null;
                        int win = 0;
                        int loss = 0;
                        StringTokenizer tokenizer = new StringTokenizer(line,",\n");
                        while(tokenizer.hasMoreTokens()) {
                              team = tokenizer.nextToken();
                              win = new Integer(tokenizer.nextToken());
                              loss = new Integer(tokenizer.nextToken());
                              }
                        Sports baseball = new Baseball(team,win,loss);
                        baseballList.add(baseball);
                        }
                  file.close();
                  }
            catch(FileNotFoundException exception) {
                  exception.printStackTrace();
                  }
            catch(IOException exception) {
                  exception.printStackTrace();
                  }
            }

      public Iterator<Sports> iterator() {
            return new BaseballIterator(baseballList);
            }
      }
```

Listing 11: The revised BaseballTeams class

```
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.StringTokenizer;

public class FootballTeams implements SportsTeams {
	private static final int MAX = 32;
	private Sports[] footballList;
	private static final String footballFile = "nfl2006.csv";

	public FootballTeams() {
		buildTeamList();
		}

	public void buildTeamList() {
		footballList = new Sports[MAX];
		try {
			BufferedReader file = new BufferedReader(new FileReader(footballFile));
			String line = null;
			int current = 0;
			while((line = file.readLine()) != null) {
				String team = null;
				int win = 0;
				int loss = 0;
				int tie = 0;
				StringTokenizer tokenizer = new StringTokenizer(line,",\n");
				while(tokenizer.hasMoreTokens()) {
					team = tokenizer.nextToken();
					win = new Integer(tokenizer.nextToken());
					loss = new Integer(tokenizer.nextToken());
					tie = new Integer(tokenizer.nextToken());
					}
				Sports football = new Football(team,win,loss,tie);
				if(current < MAX) {
					footballList[current] = football;
					++current;
					}
				else {
					System.err.println("--> ERROR: Array is full!");
					}
				}
			file.close();
			}
		catch(FileNotFoundException exception) {
			exception.printStackTrace();
			}
		catch(IOException exception) {
			exception.printStackTrace();
			}
		}

	public Iterator<Sports> iterator() {
		return new FootballIterator(footballList);
		}
	}
```

Listing 12: The revised FootballTeams class

In both Listings 11 and 12 the **iterator()** method returns an appropriate concrete iterator. In Listing 11 the method returns a **BaseballIterator**, and in Listing 12 the method returns a **FootballIterator**. Using the Iterator pattern, the code buries the ugly details (Baseball versus football? List versus array?) beneath the surface.

The original **Sports**, **Baseball**, and **Football** classes (Listings 1 through 3) don't change at all when you adopt the Iterator Design pattern. But the client application (formerly Listing 6) changes a bit. The revised client application is in Listing 13:

```
public class SportsApp {
      public static void main(String[] args) {
            Iterator<Sports> iterator = null;

            // output Major League Baseball teams
            BaseballTeams baseball = new BaseballTeams();
            iterator = baseball.iterator();
            System.out.println("### Major League Baseball 2006 ###\n");
            while(iterator.hasNext()) {
                  System.out.println(iterator.next());
                  }

            System.out.println();

            // output National Football League teams
            FootballTeams football = new FootballTeams();
            iterator = football.iterator();
            System.out.println("### National Football League 2006 ###\n");
            while(iterator.hasNext()) {
                  System.out.println(iterator.next());
                  }
            }
      }
```

Listing 13: The revised SportsApp class – a client application

When you execute the code in Listing 13 (the good code) you get the output of Figure 1 -- the same as the output of Listing 6 (also known as "the bad code"). But when other developers read the code in Listing 13, they know nothing about the data structures lurking behind the scenes. Unlike Listing 6, the code of Listing 13 completely masks the implementations of **BaseballTeams** and **FootballTeams**. The masking is especially helpful when classes (such as **BaseballTeams** and **FootballTeams**) have different underlying implementations.

So What? The Java API Already Has Iterators

The Java API has built-in iterators. So why should you bother coding the **BaseballIterator** and **FootballIterator** classes (Listings 9 and 10)? The simplest answer is, not all aggregates have iterators. For example, in this article's sports application, the **ArrayList** (in **BaseballTeams**) has a built-in iterator, but the **Sports** array (in **FootballTeams**) does not. And what if you need more functionality? The Java API has a **ListIterator** (a bi-directional iterator with **next()** and **previous()** methods) but maybe your customized iterator skips records with missing values, hiding such records from your developers downstream. For many problems, the Iterator design pattern is both the most useful and the most elegant solution.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software

http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2

Head First Design Patterns
http://www.oreilly.com/catalog/hfdesignpat/
Eric & Elisabeth Freeman
ISBN 0-596-00712-4

Object-Oriented Software Construction
http://vig.prenhall.com/catalog/academic/product/0,1144,0136291554.html,00.html
Bertrand Meyer
ISBN 0-13-629155-4

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including *Java For Dummies* and *Ruby on Rails For Dummies*, both published by Wiley.

Michael P. Redlich is a currently a Senior Research Technician at a petrochemical research organization in New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he provided technical support and developed computer telephony applications for customers. He has been a member of the Amateur Computer Group of New Jersey (ACGNJ) since 1996, and currently serves on the ACGNJ Board of Directors as President of the club. Mike previously served as Secretary and has been facilitating the monthly ACGNJ Java Users Group since 2001. His technical experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, and other programming/scripting languages in both the PC and UNIX environments. Mike has co-authored a number of articles with Barry Burd for Java Boutique. He has also conducted seminars at Trenton Computer Festival (TCF) since 1998, TCF Professional Conference since 2006, and other venues including the Emerging Technologies for the Enterprise Conference, the New York Software Industry Association (NYSIA) Java Users Group, the Princeton Java Users Group, and the Capital District Java Developers Network. Mike is the co-chair of a local Science Ambassador program where he has conducted numerous science demonstrations for various elementary schools in New Jersey. Mike holds a Bachelor of Science in Computer Science from Rutgers University.

Appendix

```
New York Yankees,97,65
Toronto Blue Jays,87,75
Boston Red Sox,86,76
Baltimore Orioles,70,92
Tampa Bay Devil Rays,61,101
Minnesota Twins,96,66
Detroit Tigers,95,67
Chicago White Sox,90,72
Cleveland Indians,78,84
Kansas City Royals,62,100
```

```
Oakland A's,93,69
Los Angeles Angels,89,73
Texas Rangers,80,82
Seattle Mariners,78,84
New York Mets,97,65
Philadelphia Phillies,85,77
Atlanta Braves,79,83
Florida Marlins,78,84
Washington Nationals,71,91
St. Louis Cardinals,83,78
Houston Astros,82,80
Cincinnati Reds,80,82
Milwaukee Brewers,75,87
Pittsburgh Pirates,67,95
Chicago Cubs,66,96
San Diego Padres,88,74
Los Angeles Dodgers,88,74
San Francisco Giants,76,85
Arizona Diamondbacks,76,86
Colorado Rockies,76,86
```

Listing 14: The file `mlb2006.csv`

```
New England Patriots,12,4,0
New York Jets,10,6,0
Buffalo Bills,7,9,0
Miami Dolphins,6,10,0
Baltimore Ravens,13,3,0
Cincinnati Bengals,8,8,0
Pittsburgh Steelers,8,8,0
Cleveland Browns,4,12,0
Indianapolis Colts,12,4,0
Tennessee Titans,8,8,0
Jacksonville Jaguars,8,8,0
Houston Texans,6,10,0
San Diego Chargers,14,2,0
Kansas City Chiefs,9,7,0
Denver Broncos,9,7,0
Oakland Raiders,2,14,0
Philadelphia Eagles,10,6,0
Dallas Cowboys,9,7,0
New York Giants,8,8,0
Washington Redskins,5,11,0
Chicago Bears,13,3,0
Green Bay Packers,8,8,0
Minnesota Vikings,6,10,0
Detroit Lions,3,13,0
New Orleans Saints,10,6,0
Carolina Panthers,8,8,0
Atlanta Falcons,7,9,0
Tampa Bay Buccaneers,4,12,0
Seattle Seahawks,9,7,0
St. Louis Rams,8,8,0
San Francisco 49ers,7,9,0
Arizona Cardinals,5,11,0
```

Listing 15: The file `nfl2006.csv`