B.A.S.E.S. Author: Phil DiMarco, Editor: Mary Menges

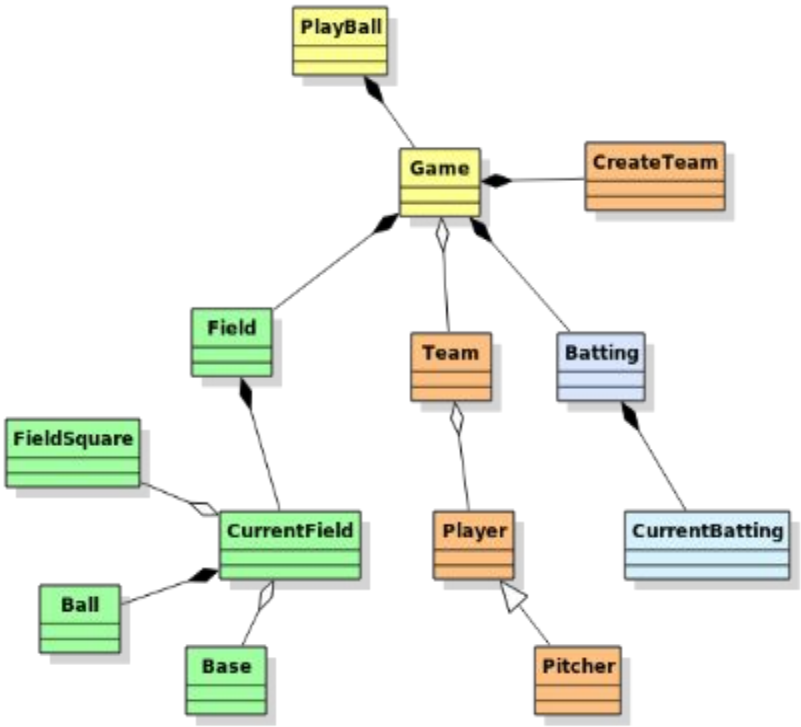
Introduction: The purpose of this document is to provide a detailed overview of the B.A.S.E.S (Baseball's Accurate Statistical Engine Simulator) program. This document will provide a high level description of the system, detailed descriptions of all classes, use of these classes in the program, and the output of the program. For a full working version of this program and it's source code see, https://github.com/reidcooper/PythonBaseballSim

High Level Description: B.A.S.E.S leverages object oriented programing, embracing the principles of data encapsulation and information hiding. This program was designed to have minimum dependence between components. The higher level components have very little dependence on the lower level components. This allows for a high degree of flexibility. This program is separated into four packages. These packages are as follows: teamPackage, battingPackage, fieldingPackage, and baseballGamePackage.

teamPackage: Responsible for the creation of team and player objects. Real team and player data is populated into these objects based on which teams the user selects to play against each other.

battingPackage: Manages the interaction between the player who is currently at bat and the opposing pitcher. This package outputs the amount of bases a player can possibly advance and whether or not an out has happened.

fieldingPackage: Manages the interaction between the players running the bases and the fielding team. This is the most complex package. It handles the structure of the



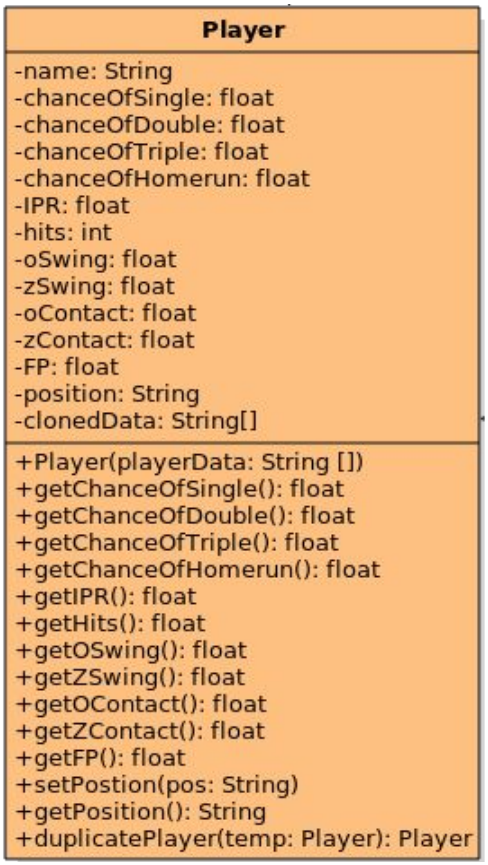
teamPackage orange, battingPackage blue, fieldingPackage green, baseballGamePackage yellow

field, the bases, and the players on base, and it manages when to mark a player as 'out'. This package outputs the amount of outs and scores that occur in the field.

baseballGamePackage: This is the highest level component of the program, since it rests on the three other packages. This package manages team switching, scores, outs, and innings. This package outputs a list of game events and the final score.

Class Level Descriptions: This section will provide a general overview of each class found in B.A.S.E.S. It will also discuss how specific components of the program work and how they interact with other components.

teamPackage: *Player*



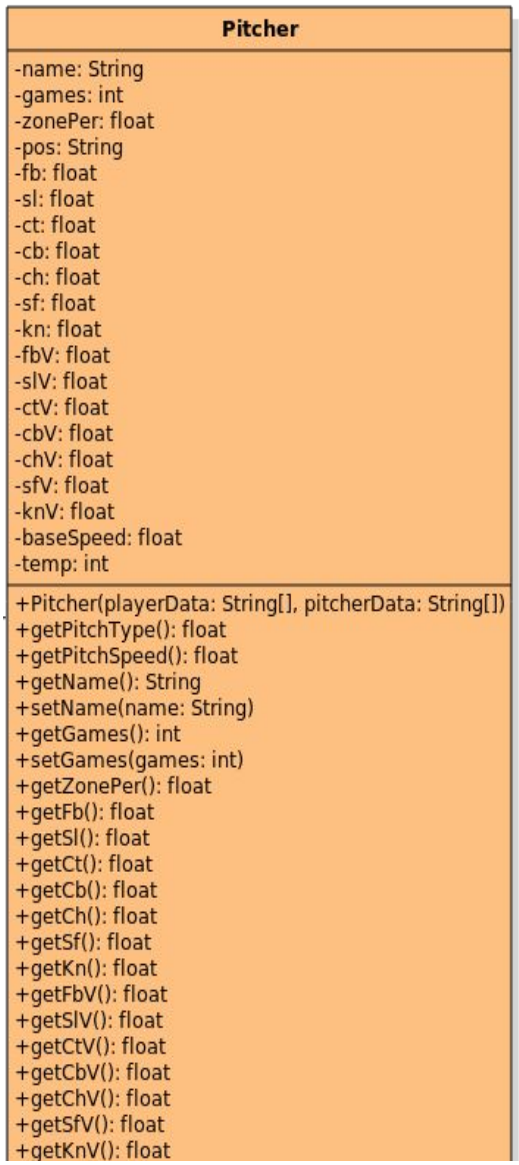
Description: This class represents a baseball player. As such this class stores all relevant information about a specific player. This includes the players name, the chance of hitting home runs, singles, doubles, triples, IPR(in play rate), fielding percentage, and the chance of a player swinging their bat given whether the pitch is in the zone or out of the zone as well as contact percent. This information is utilized heavily by the higher level components. The Player object does no computations and only stores information. The duplicatePlayer method returns a copy of the same player is was called in. This method is used only when the CreateTeam class needs to fill a fielding position that is not found on the team roster. For the program to work correctly a team needs to have one player filling every position. For a player to be created, a string array needs to be passed into the constructor at the time of creation. This information cannot be changed (except for position). The information that needs to be present in the string array is as follows. Order is also vital here.

```
1. Name          2. Position
3. Hits          4. IPR
5. O-Swing%      6. Z-Swing%
7. O-Contact%    8. Z-Contact%
9. 1B%           10. 2B%
11. 3B%          12. HR
13. FP
```

Use: The Player class should not be instantiated directly. Player objects should only be created by the CreateTeam class

teamPackage: *Pitcher*

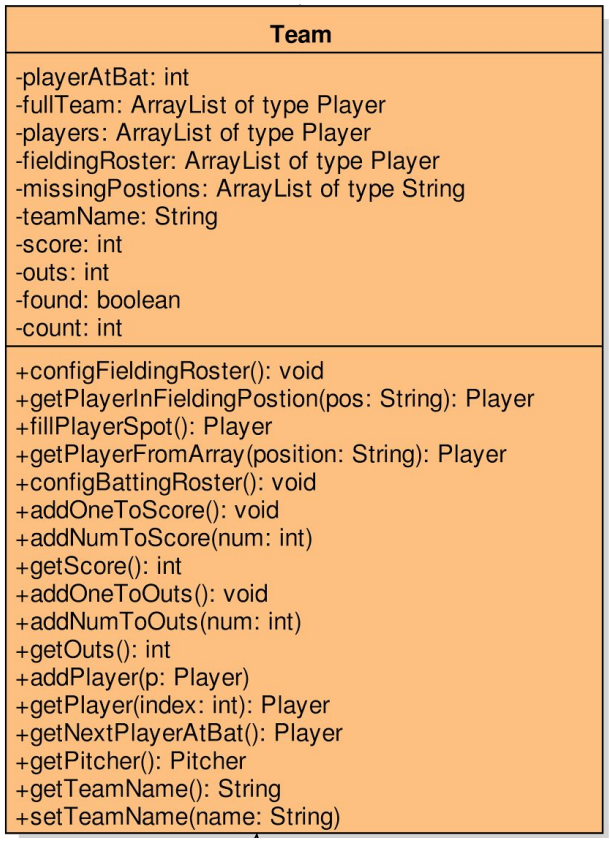
Description: This class represents a baseball Pitcher. The Pitcher class has an inheritance relationship with the Player class, where the Pitcher class is the child of the Player class. This class stores all relevant information that the program requires for pitchers. This includes the pitcher's name, zone percentage, chance of throwing a specific pitch, and the average speed at which the pitcher throws that pitch. A Pitcher object has the ability to return a random pitch speed when requested through the getPitchSpeed() method. Other than that method the Pitcher class acts in much the same way that the Player object acts in the sense that it simply just holds data about a pitcher. Like the Player class, on creation a string array must be passed into the constructor, except this time the constructor takes the player string array described above and a new string array that contains the following information.



```
1. Name                2. Games
3. position            4. Zone Percentage
5. Fastball%           6. Fastball V(elocity)
7. Slider%             8. Slider V
9. Cutters%            10. Cutters V
11. Curveball%         12. Curveball V
13. Changeup%          14. Changeup V
15. Fastball(other)%   16. Fastball(other) V
17. KnuckleBall%       18. Knuckleball V
```

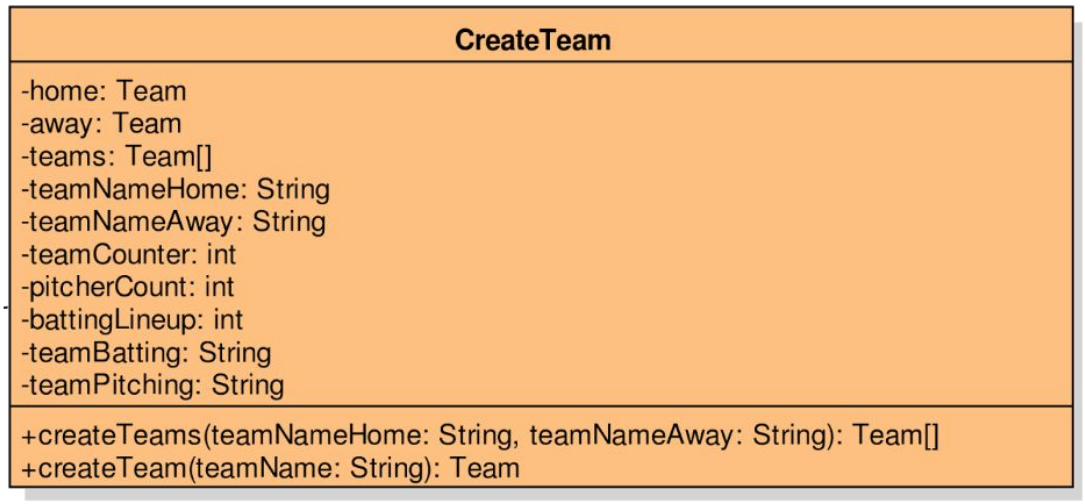
It is important to note that an empty string array of player data is passed into the super class. The Pitcher class uses inheritance because it makes the job of the Team class easier. It also makes logical sense because a pitcher is a type of baseball player.
Use: The Pitcher class should not be instantiated directly. Pitcher objects should only be created by the CreateTeam class.

teamPackage: *Team*
Description: This class represents a baseball team. It is composed of Player objects and one Pitcher object. This simulation does not allow more than one Pitcher on a team. For a future release of this software a change that would enhance the simulation would be to allow pitchers to change throughout an opposing team's at bat. The Team class is composed of three lists of Player objects and a list of strings. The first list is called fullTeam, this list contains all the players that are found in the team text file(more about this in the create team section). The players list contains the batting line up for the team, this lineup is based on the players that have played the most games. In the last position of this list is



the a Pitcher object. The next list is the fieldingRoster, this list is the most complex to construct. When the configFieldingRoster() method is called, the CreateTeam class searches through the fullTeam list and tries to pull out one of every position. This does not always work because not all teams have a player for each fielding position(for example a team might have two 2nd basemen, in such a case it is possible that a team has no player who has their official position listed as left fielder). If a position is missing from the fielding roster that position string is added to the missingPostions list, an empty player is added to the fielding roster in place. The fillPlayerSpot() method looks into the missingPostion list and takes the first player not used from the fullTeam list and creates a copy of that Player object, and changes the position to the one needed(this is where the copyPlayer() method from the player class comes in). This new Player object is then added to the fieldingRoster list at the point in the list where the empty player object is located. The next job of the Team class is to keep track of the current amount of outs a batting team has, this is handled through a getter and a method to increment the amount of outs by one. Next the Team class handles who is up next at bat. It does this by cycling through the players list. Lastly the Team class keeps track of the amount of runs a Team has scored over the course of a game.
Use: A Team object should not be instantiated directly, only the CreateTeam class should instantiate Team objects.

teamPackage: *CreateTeam*
Description: This class instantiates Team objects by transforming text files, which hold player stats, into player objects and then puts those player objects



into a Team object. This class has two Team objects, an array of Teams, which is of size two, and a string for both the home team's name and the away team's name. An array of Team objects is returned when the createTeams() method is

called. The createTeams() method is called before a baseball simulation starts, this method will instantiate the two Team objects that the user has requested. The createTeams() method calls the createTeam() method, which returns a fully created Team object, this method is called twice to populate the array of Teams. The createTeam() method pulls in player data from text files, which are stored locally. Each team has two text files (one for pitching and one for batting). The files are named as follows

```
Team_Name_pitching.txt
Team_Name_Batting.txt
Blue_Jays_pitching.txt
Mets_Batting.txt
```

A sample line from the Team_Name_batting.txt is as follows:

```
CurtisGranderson CF 128 0.65 0.262 0.663 0.627 0.85 0.617 0.211
0.016 0.156 0.996 155
```

Each ‘column’ contains a specific player stat, the stats contained in the columns are as follows:

```
1. Name          2. Position
3. Hits          4. IPR
5. O-Swing%      6. Z-Swing%
7. O-Contact%    8. Z-Contact%
9. 1B%           10. 2B%
11. 3B%          12. HR
13. FP           14. Games
```

A sample line from Team_Name_pitching.txt is as follows:

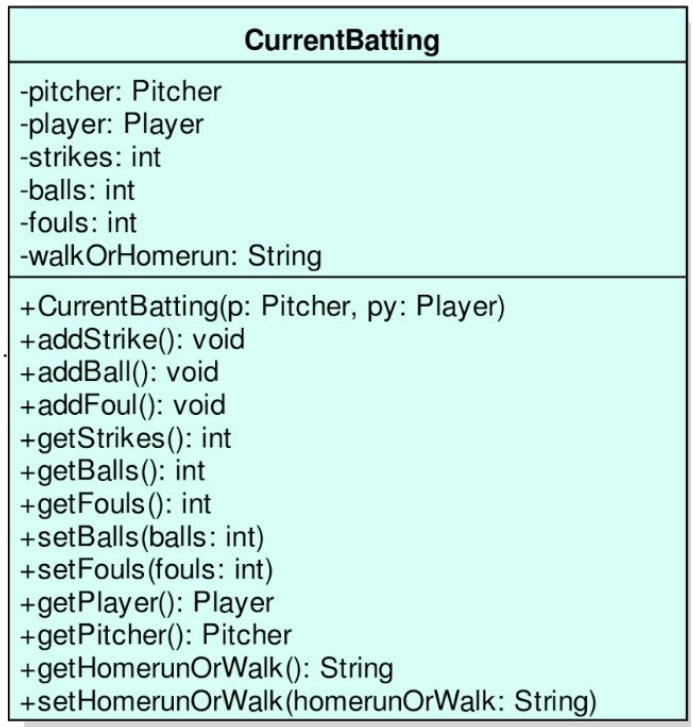
```
JeurysFamilia 76 P 0.417 0.745 96.4 0.247 86.2 0.0 0.0 0.0 0.0
0.008 89.7 0.0 0.0 0.0 0.0
```

Each ‘column’ contains a specific player stat, the stats contained in the columns are as follows:

1. Name	2. Games
3. position	4. Zone Percentage
5. Fastball%	6. Fastball V(elocity)
7. Slider%	8. Slider V
9. Cutters%	10. Cutters V
11. Curveball%	12. Curveball V
13. Changeup%	14. Changeup V
15. Fastball(other)%	16. Fastball(other) V
17. KnuckleBall%	18. Knuckleball V

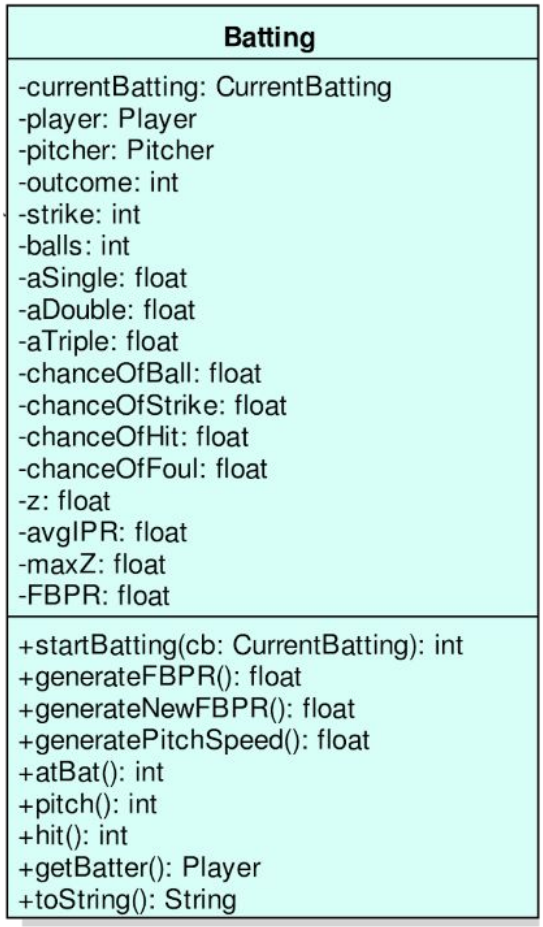
From the above information a string array for each player is constructed, this array is used in the creation of Player objects. As the Player objects are created they are added to the Team object. The createTeams() method ultimately returns an array of two teams which will be used by the baseballGamePackage to run a simulation.
Use: This class is used by the baseballGamePackage. A single instance of this class is created at the start of the program, which allows for the creation of Team objects.

battingPackage:
CurrentBatting
Description: This class stores all information related to the battingPackage. This class does no computations on data and is only responsible for setting and returning information. As such this class is mainly composed of getters and setters. These methods and attributes do not need to be discussed because of their simple nature. CurrentBatting on creation requires both a Pitcher and Player object, these objects



will be used by the Batting class to handle the interaction that will happen between the batter and the pitcher.
Use: This class is instantiated every time a new player is up at bat. This class must be passed as a parameter to the Batting class when a new player is up at bat.

battingPackage: *Batting*
Description: This class is responsible for managing the interaction between the pitcher and the batter. This class outputs the amount of bases a Player object can move(0-4). This class requires an instance of CurrentBatting. When the startBatting() method is called a new instance of CurrentBatting is passed into the class, the method sets all attributes in the class. At this point in the program the probabilities of a strike, foul, ball, and hit are generated for a specific player based on stats of both the player and the pitcher. The chance of a hit and foul change based on the speed at which a pitch is thrown. This pitch speed is decided by the Pitcher object. Inside the startBatting() method the atBat() method is called, this is where the main interaction between the batter and the pitcher takes place. A batter's distribution of hits, balls, fouls, and strikes represent a percentage, each out of 1000. Below is an example of a player's distribution of hits, balls, fouls, strikes:

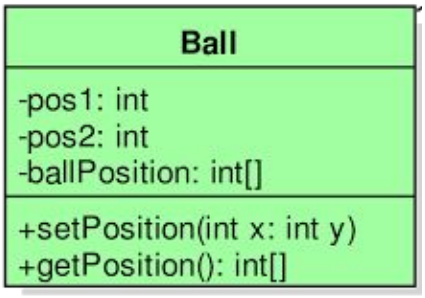



```
Chance of ball: 405.349
Chance of hit: 208.893905297
Chance of foul: 156.836238703
Chance of strike: 228.920856
```

In the atBat() method, as long as the player currently batting has less than 2 strikes, a number between 0 and 999 is generated by the pitch() method, which is called each time the atBat() method determines that a new pitch has to be thrown. This randomly generated number acts like a pitch in a real baseball game. Since all four of the above numbers add up to 1000, the numbers can be added together so for example if the number 501 is generated the program determines that a hit has happened. 405 + 208 = 612, so if a number greater than 405 and less than 612 is generated the atBat() method knows that a hit has happened. The other cases are less interesting because either the player has stuck out or they are pitched another ball. In the case of a hit another method called hit() is called. This method determines how many bases a player may be able to move, the final output of the program. Note that if a player strikes out the startBatting() method returns a zero. The hit() method works the same way as the pitch() method in determining how many bases that a player may be able to move, these are not calculated values and come directly from the batter's stats.
Use: This class is used by the baseballGamePackage. Only one instance of this class is needed. This class is used by calling the startBatting() method and passing in an instance of CurrentBatting. This class can be used with just the teamPackage and the other member of the battingPackage, CurrentBatting. Calling the startBatting() method will return an integer value between 0 and 4. This can be used to analyze the expected performance of any batter and pitcher. In the context of B.A.S.E.S this class represents the key component that allows the simulation to simulate the batting side of the program. The fielding side of the simulation requires the output of a Batting object.

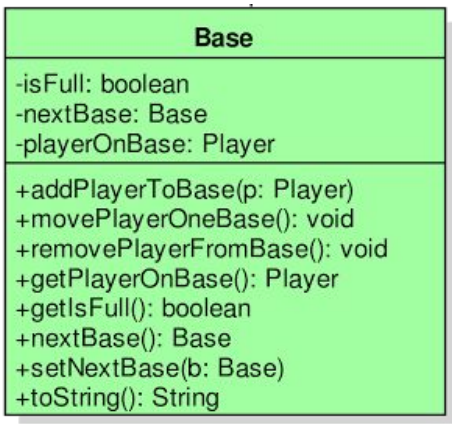
fieldingPackage: *Ball*

Description: This class is the simplest component of the fieldingPackage. This class represents a baseball and stores the current position of the ball in the field. It was decided that it was better to make the ball its own object instead of simply storing the location of the ball as an attribute of one of the other classes in the fieldingPackage, this fits better into the object oriented approach of B.A.S.E.S.



Use: A Ball object should not be instantiated directly. The ball object is used by the CurrentField class.

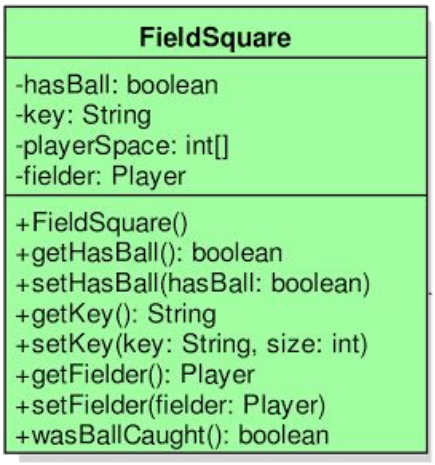
fieldingPackage: *Base*



Description: This class represents the actual bases on a baseball diamond and stores the current Player who is on base as well as the next Base a Player can move to. The Bases class should be implemented as a linked list of Base objects. The corresponding methods in this class do exactly as they suggest and do not need further explanation. This class does not manage moving players around bases, but a single base object can move a Player to the next base and remove a Player from itself.

Use: A base should not be used by itself and is supposed to be used only with other bases. CurrentField is the only class that should use Base objects.

fieldingPackage: *FieldSquare*

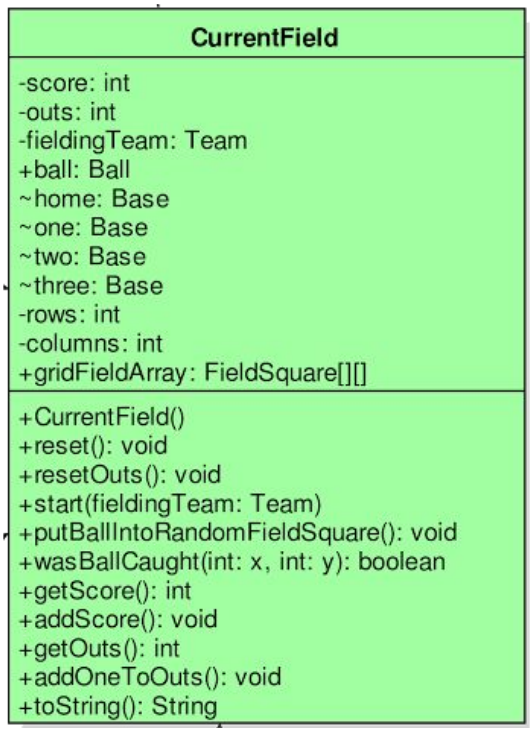


Description: This class represents the underlying structure of a baseball field. The idea behind the FieldSquare class was to turn a baseball field into a matrix, inside the squares there can be empty space, or a Player object. If there is a Player that FieldSquare has another space, called playerSpace(an array of integers, where each spot in the array represents the chance of a ball being caught by that player). If a ball 'lands' inside a FieldSquare with a Player, a random spot inside the playerSpace is chosen, the closer that spot is to the Player(the start of the array, index 0) the more likely it is that the player will catch the ball. If the ball lands exactly on Player, the Player's fielding percentage is used to determine if the ball is caught or not. The process of determining whether a ball was caught or not is handled by the wasBallCaught() method. If a FieldSquare will have a Player inside it, that FieldSquare must be assigned a key, which is the position of the Player to be assigned, and also the size of that respective Player's area.

Use: A FieldSquare object is only used by the CurrentField class. FieldSquare is not a standalone class and should only be instantiated inside a matrix structure.

fieldingPackage: *CurrentField*

Description: This class acts in much the same way as CurrentBatting in the sense that it stores data rather than computing anything. This class intends to pull all the above elements talked about in the fieldingPackage into a single unified baseball field structure. This class keeps track of outs and scores that happen on the field. It holds both the Base (note the Base objects in this class are protected not private to allow the Field class easier access) and FieldSquare structures talked about above and a reference to the fielding Team object. The FieldSquare matrix has the following structure:



FieldSquare Array (x marks plate, P is pitcher square)

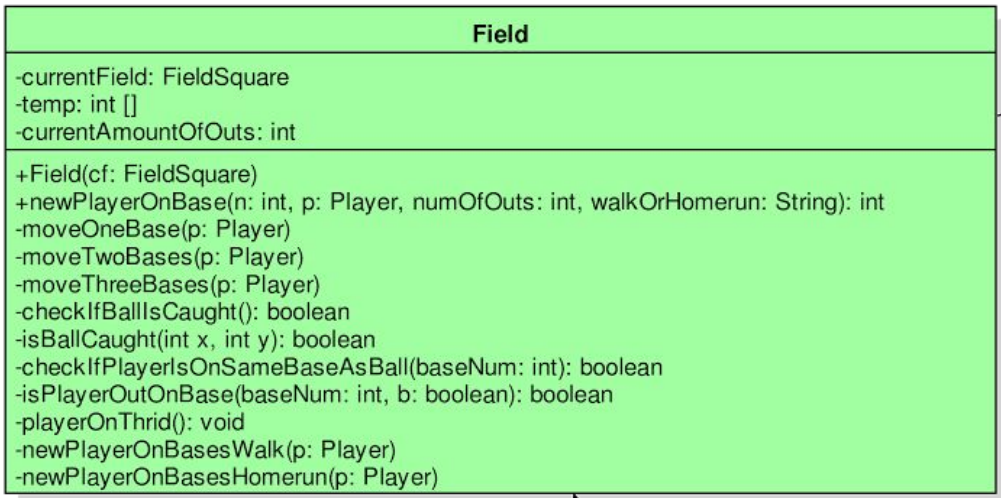
```
_____________________________
| 0        | 1   |   2       |
_____________________________
| 3        | 4x  |   5       |
_____________________________
| 6x       | 7P  |   8x      |
_____________________________
|          | 9x  |           |
-----------------------------
```

FieldSqure Positions

```
gridFieldArray[0][0].setKey("LF", 15)
gridFieldArray[0][2].setKey("CF", 15)
gridFieldArray[0][4].setKey("RF", 15)
gridFieldArray[2][1].setKey("SS", 8)
gridFieldArray[2][2].setKey("P", 8)
gridFieldArray[2][3].setKey("2B", 8)
gridFieldArray[3][0].setKey("3B", 8)
gridFieldArray[3][2].setKey("C", 8)
gridFieldArray[3][4].setKey("1B", 5)
```

The start() method needs to be called when a new team is up at bat. This method resets the field and changes the fielding team. The wasBallCaught() method, calls the respective wasBallCaught() method in the FieldSquare object that the ball was hit into. The putBallIntoRandomFieldSquare() method is called each time the ball is hit into the field by a batter.
Use: CurrentField must be used in conjunction with the Field class. Unlike the CurrentBatting, a new CurrentField object does not have to be created every time a new Team object enters the field. By calling the start() method and passing the Team object that is fielding, the CurrentField object is ready to be used by the Field class.

fieldingPackage: *Field*
Description: This class utilizes the data stored in CurrentField to produce the interactions that are likely to occur between the fielding team and the baserunners. This class represents what a person might see when watching a baseball game after the batter has hit the ball in the field. This class handles when the ball is caught by the fielding team, when a baserunner gets out, moving the baserunners around the bases appropriately, and incrementing scores and



outs. The Field class tightly integrates all above components of the fieldingPackage in order to achieve this. The key method in this class is newPlayerOnBase(), this method is called every time a new player needs to be put on base. This method is passed the Player object who needs to be put on base, the number of bases they can move and if the hit was a special case(meaning it was either a homerun or a walk). This method returns the

number of outs that have happened on the during the fielding process. The addNewPlayerToBase() method works in the following way

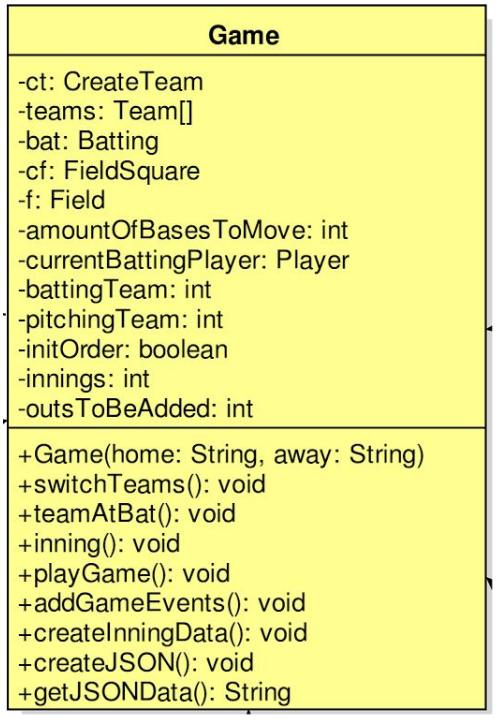
1. Check for special case of homerun or walk, if one of these cases has happened call the appropriate newPlayerOnBasesHomerun() method or newPlayerOnBasesWalk() method
2. Put the ball into a random FieldSquare by calling putBallIntoRandomFieldSquare() method from the CurrentField object
3. Check if the ball was caught by calling checkIfBallWasCaught() method, if the ball was caught at caught the runner may run the amount of bases their hit allows, this is where a try ands soon as a team reaches three outs, an error gets thrown as soon as a team reaches three outs.
4. Before a player can advance a base, the isPlayerOutOnBase() method must be called, this method takes the base number the player is moving to, and also a boolean which is generated from calling the method checkIfPlayerAndBallAreOnSameBase().
    a. `isPlayerOutOnBase(3, checkIfPlayerIsOnSameBaseAsBall(3))`, In this example a player is moving to third base
        i. checkIfPlayerIsOnSameBaseAsBall() returns true if the ball has been caught causing the player to get out, the FieldSquare object pertaining to the specific base determins if a ball has been caught or not
    b. The isPlayerOutOnBase() method returns a boolean, if the player is out it return true
    c. The isPlayerOutOnBase() method will throw an error if a the running team has 3 outs, this error is caught by newPlayerOnBases() method
5. The logic for advancing players around bases will not be discussed here, but each case for players moving around the bases is thoroughly accounted

```
for. To see how this logic works go to the github
link at the top of this document and read the
source code for this program.
```

Use: The Field class is used by calling the addNewPlayerToBases() method. There should only be one instance of this class made. A Fielding object must be used with a CurrentField object.

baseballGamePackage: *Game*

Description: The game class represents a complete baseball game from start to finish. This class communicates with the all the lower level packages in order to simulate a baseball game. This class integrates team creation, batting, and fielding into a single baseball game, and outputs a JSON files of all game events that transpired throughout the simulation. To show how all the classes described in this document can work together to simulate a baseball game, a walk through of teamAtBat() method is necessary. This method is the core of the baseball game simulation and it is run many times throughout the course of a simulation. The next section will consist of a line by line analysis of the teamAtBat() method




```
cf.start(teams[pitchingTeam])
#line 1
```

```
This line sets up the CurrentField object by filling all
player references with pitching team thus preparing the
field for the batting team.  Note that the variable
pitchingTeam is an integer between the number 0 and 1, this
number is the index of the team whose turn it is to field.
This works because teams[] is an array of size 2 and it is
populated with two teams.  When it it time for teams to
switch the numbers that pitchingTeam holds goes from 0 to 1
or 1 to 0.  The variable battingTeam works the same way.
```

```
while(teams[battingTeam].getOuts() < 3)
```
#Line 2

This line starts a loop that keeps the current at bat team up at bat until they accumulate three outs, the outs accumulated are stored in the Team object, that is why the getOuts() method is called on the currently batting team.

```
currentBattingPlayer = teams[battingTeam].getNextPlayerAtBat()
```
#Line 3

This line grabs the player who is currently going to be up at bat, the Team object knows which team member needs to be up at bat.

```
CurrentBatting cb = new CurrentBatting(teams[pitchingTeam].getPitcher(),
currentBattingPlayer)
```
#Line 4
```
amountOfBasesToMove = bat.startBatting(cb);
```
#Line 5

line 4 creates a new instance of CurrentBatting, passing in the correct batter and pitcher who need to be up. The CurrentBatting object needs to hold a reference to these players, this object is needed by the Batting object to facilitate the interaction between the batter and the pitcher. In line 5 the method startBatting() is called passing in the CurrentBatting created above. Note the object bat is an instance of Batting. The startBatting() method returns the amount of bases a player can move after being pitched the ball, the result is stored in the amountOfBases to move variable.

```
if(amountOfBasesToMove > 0 ){
```
#Line 6

This line checks to see if the batter has had an interaction with the pitcher that allows him to advance any bases (did he hit the ball or did he strike out). If this condition is true then the batter needs to become a runner and enter the field, if this condition is false the batter needs to be marked as out, and an out needs to be added to the batting team.

```
outsToBeAdded = f.newPlayerOnBases(amountOfBasesToMove,
currentBattingPlayer, teams[battingTeam].getOuts(), cb.getHomerunOrWalk())
#Line 7
teams[battingTeam].addNumToOuts(outsToBeAdded)
#Line 8
```

Line 7 executes if the above if statement is true. If the above statement is true it means that the player has successfully hit the ball and is ready to run the bases. This happens by calling the newPlayerOnBases() method and passing into that method the amount of bases he is going to move, the Player object who was at bat, the amount of outs the batting team has, and if the hit was a walk or homerun, which is stored in the CurrentBatting object. This method will run all the players around the bases and return the amount of outs that have happened on the field. Note f is a Field object and cb is a CurrentBatting object. Line 8 adds all the outs that have happened in the field to the respective Team's amount of outs.

```
else{teams[battingTeam].addOneToOuts()}
#Line 9
```

The above line is the else statement that closes off the if statement that started in line 6. This statement executes if the Batting object returns that the player can advance 0 bases. This means that the player has struck out. In this case an out needs to be added to the batting team.

```
teams[battingTeam].addNumToScore(cf.getScore());
#Line 10
teams[battingTeam].setOutsToZero();
#Line 11
cf.reset();
#Line 12
```

The last three lines of this method execute after a team has finished their at bat and accumulated 3 outs(the initial while loop has stopped executing). Line 10 adds the runs a team has scored to the collective team score. Line 11 resets the batting team's outs to 0. Line 12 resets the CurrentField object to its initial state of

```
having no players on base.  These two lines set all things
back to how they should be so that when the next team goes
to bat all things wont get messed up.
```

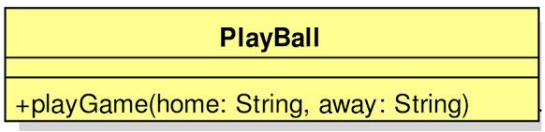
The atBat() method described above represents a half inning. The rest of the Game class simply executes the atBat() method in such a way that a baseball game is simulated.

Use: The Game class is used by creating an instance and passing in the two teams that the user wants to player each other into the constructor, and then calling the playGame() method. There can be as many Game objects created as a user wishes. This Game object is basically a fully functioning baseball game simulation. To output the results of the game to a JSON file, call the getJSONData() method. Below is a list of all the names of the teams who can play each other in the simulation, the strings must be entered exactly as they appear below.

```
Yankees, Phillies, Red-Sox, Angels, White-Sox, Cubs,
Mets, Giants, Twins, Tigers, Cardinals, Dodgers,
Rangers, Rockies, Braves, Mariners, Brewers, Orioles,
Reds, Astros, Athletics, Nationals, Blue-Jays,
Marlins, Diamondbacks, Indians, Padres, Pirates, Rays,
Royals
```

baseballGamePackage: *PlayBall*

Description: This class simply contains one method to start a baseball simulation, it just requires two strings which are the teams who will be playing each other.



B.A.S.E.S. Output: This program can output all game events that happened during a simulation to a JSON file. This section will describe how to create the output, the naming conventions of the output file, and finally the contents of the JSON file.

Creation of JSON File: To create the JSON file output simply call the getJSONData() method on any Game object after a simulation has finished running

Name of JSON File: The outputted JSON file is named by a timestamp followed by the teams who played each other. The following is how the file name is formated

Year-Month-Day-Hours-Minutes-Seconds_Team-One_Team-Two_.json

Below are a few examples of the file names

```
2015-07-24-14-58-42_Braves_Cubs_.json
2015-08-10-10-41-59_Red-Sox_Blue-Jays_.json
2015-08-04-10-42-20_Mets_Yankees_.json
```

JSON File Contents: The JSON file contents are key to communicating the results of B.A.S.E.S. to the world. The JSON file is separated into three sections, the first section represents the inning number, the second section is an event number, and the third section is an event code and a description of that. For example to get the first the description of the first event of the first inning you could type [0][0]["description"] and the result would print “The Braves are up at bat!”(see example JSON text below)


```
{
      "code": "START-HALF-INNING",
      "description": "The Braves are up at bat!"
},
{
      "code": "NEW-BATTER",
      "description": "EvanGattis is next up to bat!"
},
{
      "code": "BALL",
      "description": "Ball 1"
},
{
      "code": "STRIKE",
      "description": "Strike 1"
}
```

Codes: The codes used above all correspond to a game event. Below is a listing of all of the game events that B.A.S.E.S. will output

START-HALF-INNING - the start of a new half inning
NEW-BATTER- a new batter is up at bat
BALL - a batter got a ball
STRIKE - a batter got a strike
FOUL-STRIKE - a batter got a foul that was a strike
FOUL - a batter got a foul
BB - a batter walked
KO - a batter struck out
1B - a batter hit a single
2B - a batter hit a double
3B - a batter hit a triple
HR - a batter hit a homerun
RUN-SCORES - a runner scored
OUT-BH - the ball was caught
OUT-1BH - the hitter got thrown out at first
OUT-2BH - the hitter got thrown out at second
OUT-3BH - the hitter got thrown out at third
OUT-2B - the runner got out while running to second
OUT-3B - the runner got out while running to third
OUT-HP - the runner got out while running to home plate
DIAMOND - tells which bases are occupied by a base runner and which bases are empty, 000 in the description means all bases are empty, and 111 means all bases are full. 010 means there is a runner on second base, and so on.