**Software Life Cycle Blueprint for College Voting Application**

**Drafted by Robin A. and Zach D.**

Problem Specification Step

The purpose of the problem specification step is to give clarity and draft out a more concise version of the problem, giving an algorithm that can be used by the programmers to achieve a functionality that cannot be easily misinterpreted. This can change a vague problem that can be interpreted infinite ways into an algorithm that will be able to be translated directly into code, exactly as intended. Ways this can be accomplished is by receiving further clarification from the client over wording that may come off as ambiguous, and drafting more concise instructions using formal and direct language. By direct collaboration with the client and the design team, a draft can be achieved that can be understandable and to the specifications of the client, while also being clear enough to remove much of the room for error that could come from vague wording.

Project Specification Document

Due to the four segmented districts, it is assumed that each district will have different candidates for each, so giving the option to plug in the names of each of the candidates into each district will be important to the program. Due to this, it would be best to take an object-oriented approach to coding the program, as to allow the same manipulations and tallies for voting, alongside printing of the candidate names, be done regardless of how many districts there are, or if the candidates are similar or different. While they state that there will be between 4-10 candidates, there is no real need to worry about the lower bounds; allowing them to define the number of candidates and dynamically creating an array to that value will be sufficient (this allows reuse if candidate number changes). We can also assume that individuals will be monitoring the voting, so it is unnecessary to put in checks or prompts for student ID’s to make sure students don’t vote more than once, especially since the only way to implement this would be to make sure the school has a database of each student ID easily accessible and ready to plug in to our program, and without this information, we can’t design the project based around the assumptions that they would. We are also given no information as to how long the voting process will take, so instead of worrying about adding in a time prompt to vote, nor taking the system time, it would be best to have the voting official set up a key on launch that, if it is typed in place of a valid district, will end the voting process and tally up the results.

Program Design Step:

The program design step in the software life cycle methodology focuses mainly on fleshing out what is absolutely necessary to include in the code of the project at hand. By determining what the code should consist of for basic functionality, the developer can outline how a program should flow before any code is written. This can be very helpful in creating code that is easily readable, and that makes sense chronologically. Additionally, since the “pseudocode” has been broken down into less complex tasks, writing the actual code can be done in a step-by-step process, which can further expedite the coding and debugging phase of the cycle.

Program Design Document:

Our local college is requiring voting software for the upcoming election. The college is branched out into four distinct districts, so each district should be handled separately within the voting software (perhaps a different object of the same class to represent each of the four districts). Within each of these four districts, 4-10 candidates will be up for election/reelection, so this District class should be able to receive a String array holding the names of each of the candidates on instantiation. Beyond that, each district should define an integer on instantiation that can be used to handle each of the incoming votes. On program launch, the initial launcher of the program, which would be an election official, will first be prompted to type in a numeric key to be used to end the voting process when necessary. This will be accomplished using a scanner storing a valid selection into an integer variable. Next, the voting official need to be prompted to say how many districts will be in this upcoming election. This value will be stored as an integer, no less than 1. Afterwards, a prompt asking how many candidates will be up for election in each district (use a loop that iterates for each district) which will be stored as a variable. Within the same loop, using the candidate amount value given, the user will be prompted to list the names of each candidate, first name then last name, which will be stored in a String array created from the numeric amount of candidates given. Once this is done, a District object will be created, passing the candidateName array as parameters. During instantiation of each object, the passed array will be sorted through and used to populate the candidates array which will be part of the District class. These districts will be stored in a District object array, where district[0] will coincide with district 1, etc. A UML diagram for the District object is listed below.

|  |
| --- |
| District |
| -candidates : String[]  -candidateVote: int[10] |
| +District(String[])  +setCandidates(String[]): void  +displayCandidates() void  +addVote(int): void  +sortByVotes(): void  +toString(): string |

Once voting has been set up, an integer variable is created that will be used for a while loop that will run throughout the voting process, which will only stop when the value of the user’s integer variable is the same as the integer key the voting official set up on start. Within this while loops, the user will first be given the prompt to choose which voting district they are from, which will allow the values of the amount of districts created by checking the length of the district array (from 1 to district.length) alongside the value of the voting official’s key, which will end the loop and move on to tallying. Once a valid district is chosen, the names of the candidates will print in alphabetical order, ascending via last name, preceding with a numeric value (e.g. “1. John Jackson”), and a second prompt will be given asking for the user to select a candidate via their corresponding number. If a valid integer is entered, this number will be passed to the addVote method of the selected District object, which will access the candidateVote array at the specified index minus 1 (candidate 1 would be index 0 etc) and increment the value of that integer value by one. After the voting loop breaks, the candidateVote int array should be sorted in descending format, swapping the matching candidate values at the same time to keep consistency. A for loop will call the toString methods of each District object, which will be formatted to print the names of the candidates from top to bottom of the array (which should be sorted by this point to print from most to least votes).

Selection of Algorithm/Data Structures Step:

The algorithm/data structure portion of the SLC is used to help select which algorithms should/will be implemented in the upcoming project, alongside selecting data structures that will work best with the task on hand. While specific data structures haven’t been touched on too much as of yet, nor which data structure works best for certain jobs, in a real-life scenario, having an idea on which ones to select to help with optimization/speed is the main reason as to why this is such a critical step.

Algorithms and data structures used:

For this program, we will keep it rather basic by sticking to arrays for the data structure, since we should never need to handle more than 10 pieces of data at any time. For the sorting algorithm, again due to the smaller scope of the data being handled, a simple insertion sort should suffice for our needs.

Coding/Debugging step

The coding/debugging step is one of the more “straightforward” steps, in the sense that it’s the primary step that we’ve been doing with all of our trivial projects. The purpose of this step is to take the blueprint and design choices of the previous steps and translate it into working, executable code.

Source Code for voting software

/\*\*

\* District class; allows districts to be created for the sake of candidate voting by district

\* @author Zach D. & Robin A.

\*/

import java.util.Scanner;

public class District {

private String [] candidates;

private Integer [] candidateVote;

/\*\*

\* Constructor; uses an array of candidates passed by voting official

\* @param newCans - String array of candidates passed by voting official on startup

\*/

public District(String[] newCans) {

setCandidates(newCans);

sortCandidates();

}

/\*\*

\* Prints out a list of all candidates; for use in showing voters all candidates

\*/

public void displayCandidates () {

int i = 0;

for (String j: candidates) {

System.*out*.printf("Candidate #%s: %s%n",i+1, j);

i++;

}

}

/\*\*

\* Returns length of candidate array for use of setting a voting parameter (valid vote between 1 and

\* candidate.length)

\* @return - length of candidates Array

\*/

public int candAmount () {

return candidates.length;

}

/\*\*

\* Adds a vote to the vote array to line up with the coinciding candidate's index

\* @param newVote

\*/

public void addVote (int newVote) {

candidateVote[newVote-1]++;

}

/\*\*

\* Initializes the candidates array and vote tally array using the candidates passed from

\* the creation of the District class

\* @param newCans - Array of all the candidates for this district

\*/

private void setCandidates (String [] newCans) {

candidates = newCans;

candidateVote = new Integer[newCans.length];

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

candidateVote[i] = 0;

}

}

/\*\*

\* Sorts the candidate array by last name, ascending order, using an insertion sort.

\*/

private void sortCandidates() {

String nameTemp = "";

for (int i = 1; i < candidates.length; i++) {

for (int j = i; j > 0; j--) {

String compA = candidates[j-1];

Scanner parseName = new Scanner(compA);

compA = compA.replace(parseName.next() ,"").trim();

String compB = candidates[j];

parseName = new Scanner(compB);

compB = compB.replace(parseName.next(), "").trim();

if (compA.compareToIgnoreCase(compB) > 0) {

nameTemp = candidates[j];

candidates[j] = candidates[j-1];

candidates[j-1] = nameTemp;

}

}

}

}

/\* (non-Javadoc)

\* @see java.lang.Object#toString()

\* Returns the voting results alongside the names of each candidate

\*/

*@Override*

public String toString() {

String voteTally = "";

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

voteTally = voteTally.concat(String.*format*("Candidate #%s: %s. Total votes: %s%n", i+1,

candidates[i], candidateVote[i]));

}

return voteTally;

}

}

/\*\*

\* Voting class used in conjunction with District objects to create a voting booth that can

\* handle multiple districts with different candidates in each district. There's no security

\* features implemented, due to the assumptions that voting officials will be there to monitor

\* the process, nor were security features requested

\* @author Zach D. & Robin A.

\*/

import java.util.InputMismatchException;

import java.util.Scanner;

public class Voting {

/\*\*

\* Main method used to create a voting environment to be used for the actual voting process.

\* @param args

\*/

public static void main(String[] args) {

int voteKey = 0;

int districtNum = 0;

District [] votingDistrict;

Scanner userInput = new Scanner (System.*in*);

// Voting key to break voting once the process is finished

System.*out*.println("Hello! If you are an authorized voting official, please enter "

+ "a positive numeric key (4 or more digits) to end the voting process once finished");

while (voteKey < 1000) {

try {

voteKey = userInput.nextInt();

if (voteKey < 1000) {

System.*out*.println("Please make sure your key is 4 (positive) digits or greater");

}

}

catch (InputMismatchException e) {

System.*out*.println("Please enter valid, positive integers only!");

userInput.nextLine();

}

}

// Allows voting official's input of # of districts

System.*out*.println("Thank you! Be sure to write down your key; it is required to end voting\nNext,

tell me how many districts will be part of this voting cycle?");

while (districtNum < 1) {

try {

districtNum = userInput.nextInt();

if (districtNum < 1) {

System.*out*.println("Districts must be greater than one!");

}

}

catch (InputMismatchException e) {

System.*out*.println("Please enter a valid, positive integer!");

userInput.nextLine();

}

}

votingDistrict = new District[districtNum];

// Creates District objects by passing an array of each district's candidates

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

int numCandidates = 0;

System.*out*.printf("How many candidates are in district #%s?%n",i + 1);

while (numCandidates < 1) {

try {

numCandidates = userInput.nextInt();

}

catch (InputMismatchException e) {

System.*out*.println("Please enter a valid, positive integer!");

userInput.nextLine();

}

}

String [] tempCandidates = new String[numCandidates];

userInput.nextLine();

for (int j = 0; j < numCandidates; j++) {

System.*out*.println("Please enter candidate #" + (j+1) +"; first name, then last");

tempCandidates[j] = userInput.nextLine();

}

votingDistrict[i] = new District(tempCandidates);

}

System.*out*.println("Thank you very much! Voting may now commense");

int userVote = 0;

int userDistrict = 0;

// Loop that allows voting from students

while (userDistrict != voteKey) {

userDistrict = 0;

userVote = 0;

System.*out*.println("Welcome! Please enter your voting district");

while (userDistrict < 1 || userDistrict > districtNum) {

try {

userDistrict = userInput.nextInt();

if (userDistrict == voteKey) {

break;

}

if (userDistrict < 1 || userDistrict > districtNum) {

System.*out*.println("Please enter a valid district only!");

}

}

catch (InputMismatchException e) {

System.*out*.println("Please enter a valid, positive integer!");

userInput.nextLine();

}

}

if (userDistrict == voteKey) {

System.*out*.println("Admin key entered. Ending voting process and tallying up votes.");

break;

}

System.*out*.println("Thank you! Here are the candidates listed for your district:");

votingDistrict[userDistrict - 1].displayCandidates();

System.*out*.println("Which candidate would you like to vote for? Enter the number next to the

candidate's name");

while (userVote < 1 || userVote > votingDistrict[userDistrict - 1].candAmount()) {

try {

userVote = userInput.nextInt();

if (userVote < 1 || userVote > votingDistrict[userDistrict - 1].candAmount()) {

System.*out*.println("Please select a valid candidate!");

}

}

catch (InputMismatchException e) {

System.*out*.println("Please enter a valid, positive integer!");

userInput.nextLine();

}

}

votingDistrict[userDistrict -1].addVote(userVote);

System.*out*.println("Thank you for voting!");

}

// Print results

districtNum = 0;

for (District j: votingDistrict) {

districtNum++;

System.*out*.printf("Results for district %s:%n%s%n", districtNum, j);

}

}

}

Testing & Verification Step

The testing and verification step of the software life cycle methodology focus mainly on the testing the written code. Upon the near competition of the code, testing it becomes absolutely necessary to ensure proper functionality. Ideally, every possible scenario should be tested to see how the code functions under those conditions, but realistically this is impossible. There are far too many situations that a program could experience to test them all. With that being said, it is imperative that all obvious outcomes are tested, and all apparent variables are verified. By testing the code, the developer can identify errors within the code, and work to resolve them in a manner that will not impede other functionality the program has.

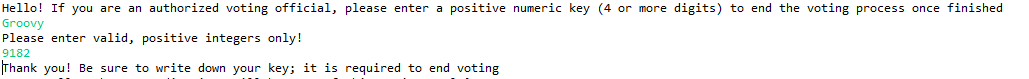
Testing & Verification Document

The following is a simple test plan of the code out group has written to record votes. Screen prints of the results are shown below the test plan.

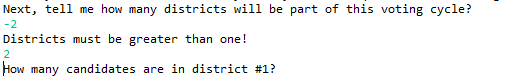
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Reason/What is being tested | Specific Input | Expected output | Actual output | Changes needed? |
| Invalid Admin code | “Groovy” | -Display error message “Valid positive integers only”  -Prompt Admin key enter again | same | n/a |
| Invalid amount of districts | -2 | -Display error message “Districts must be greater than one”  -Prompt district amount again. | same | n/a |
| Candidate names after alphabetical sort (once candidates are entered). | Randy Savage,  Kung Fury,  Danny DeVito | 1.Danny Devito  2.Kung Fury  3. Randy Savage | same | n/a |
| District index out of range during voting | -3 | -Display error message “Please enter a valid district only!”  -Prompt voter again | same | n/a |
| Candidate index out of range during voting | -1 | -Display error message “Please select a valid candidate’  -Prompt voter again. | same | n/a |
| Admin code exit, which will lead in to results | Admin code created on launch (ex. 9182) | Terminate program, print voting results; ex for candidates listed above:  Results for district #1:  #1. Danny Devito: \*amountOfVotes\*  #2. Kung Fury: \*amountOfVotes\*  #3. Randy Savage: \*amountOfVotes\* | same | n/a |

Formal Test Plan Results

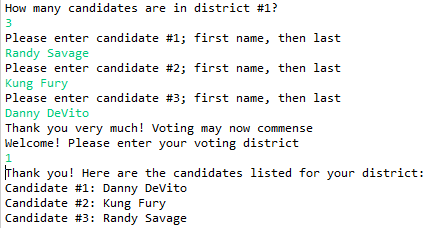
1.



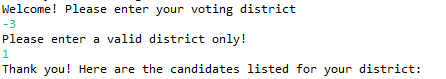
2.



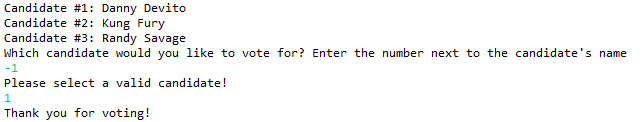
3.



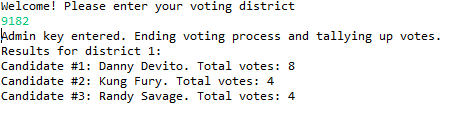
4.



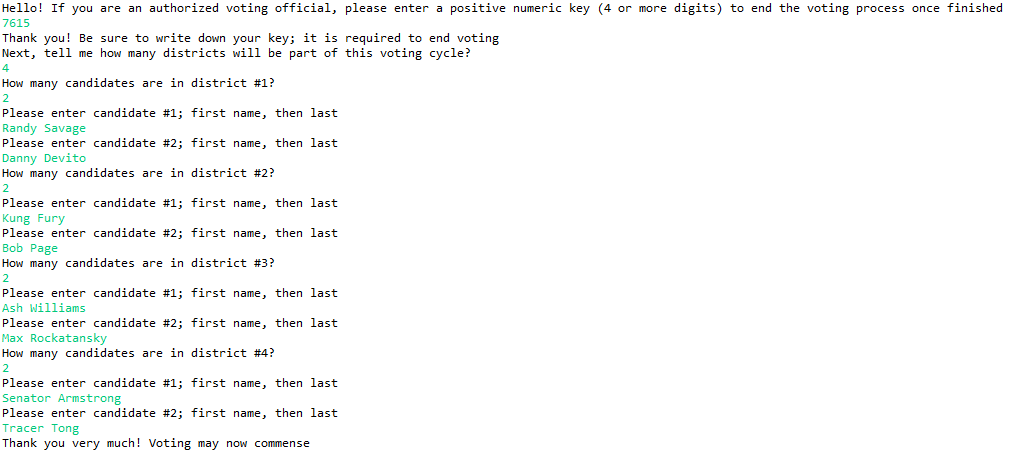
5.



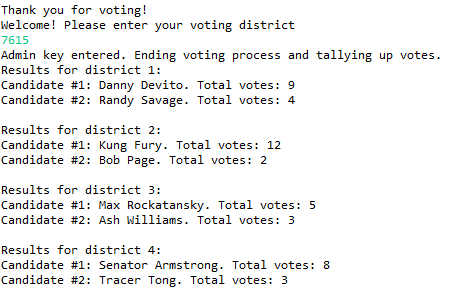
6.



Results for program w/ multiple districts & candidates



\*MOCK VOTING DONE HERE\*



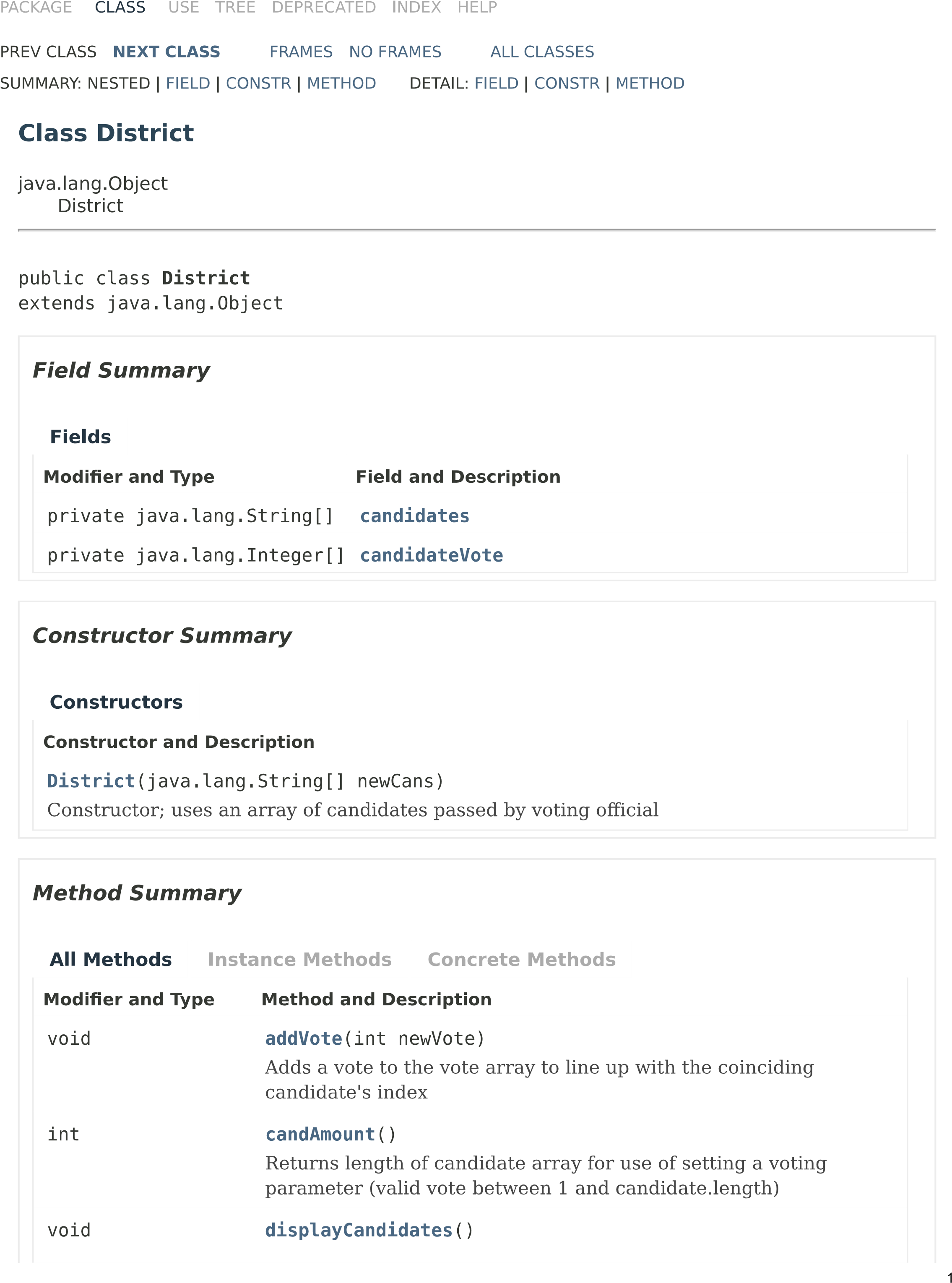
Documentation and Support Step

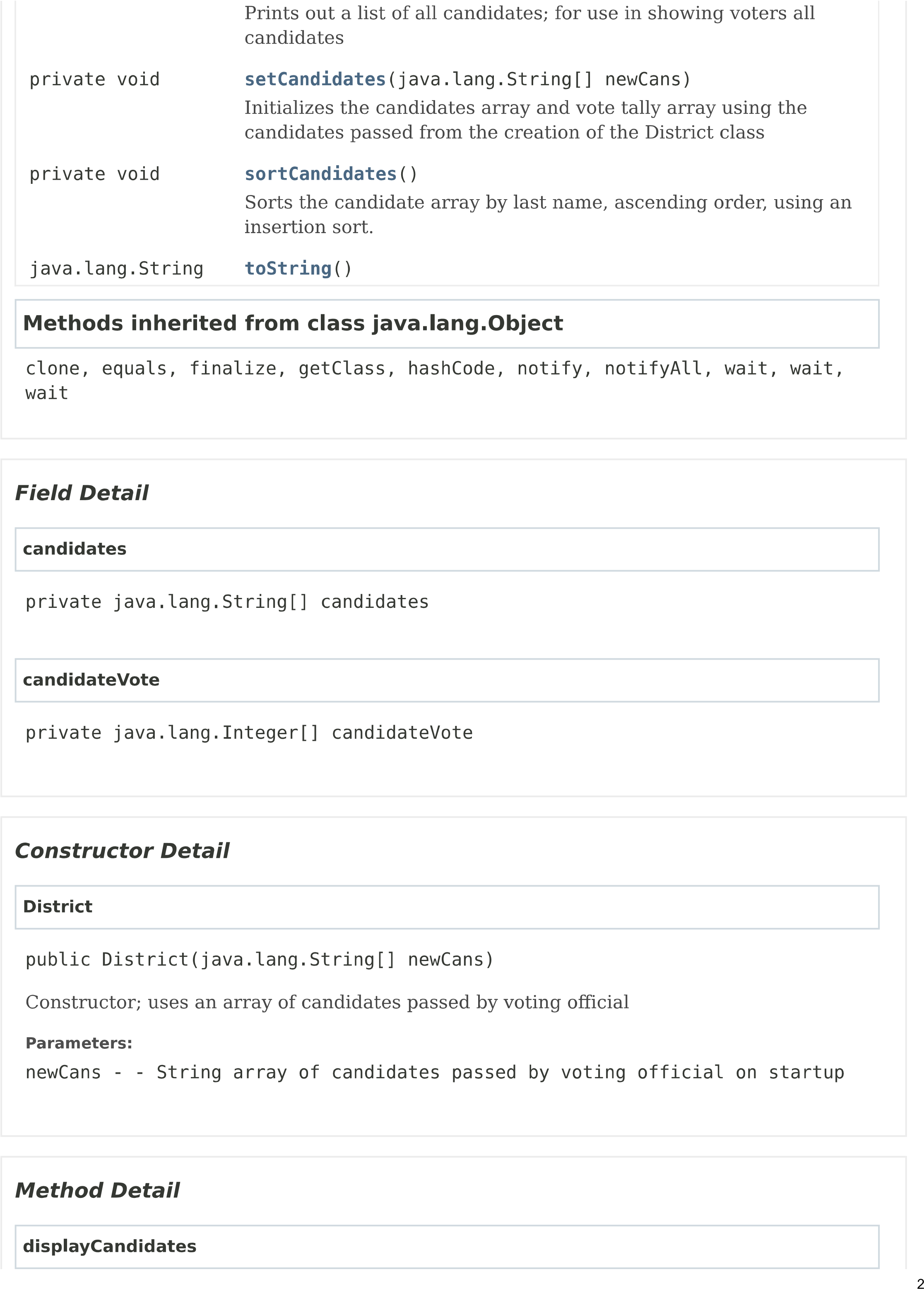
The documentation and support step in the software life cycle methodology consists of properly recording the process of creating the code in order to ensure that the code can be manipulated by multiple users that may have varying degrees of familiarization with the program. This would include creating instructions in which any user can follow to work the program, as well as a sort of guide for anyone who may update or provide assistance with the program in the form of test plans, from the testing and verification step, and API documentation. In doing so, developers guarantee that users can properly use their program, and that anyone working with the software directly understands the basic functions of the program that must be maintained.

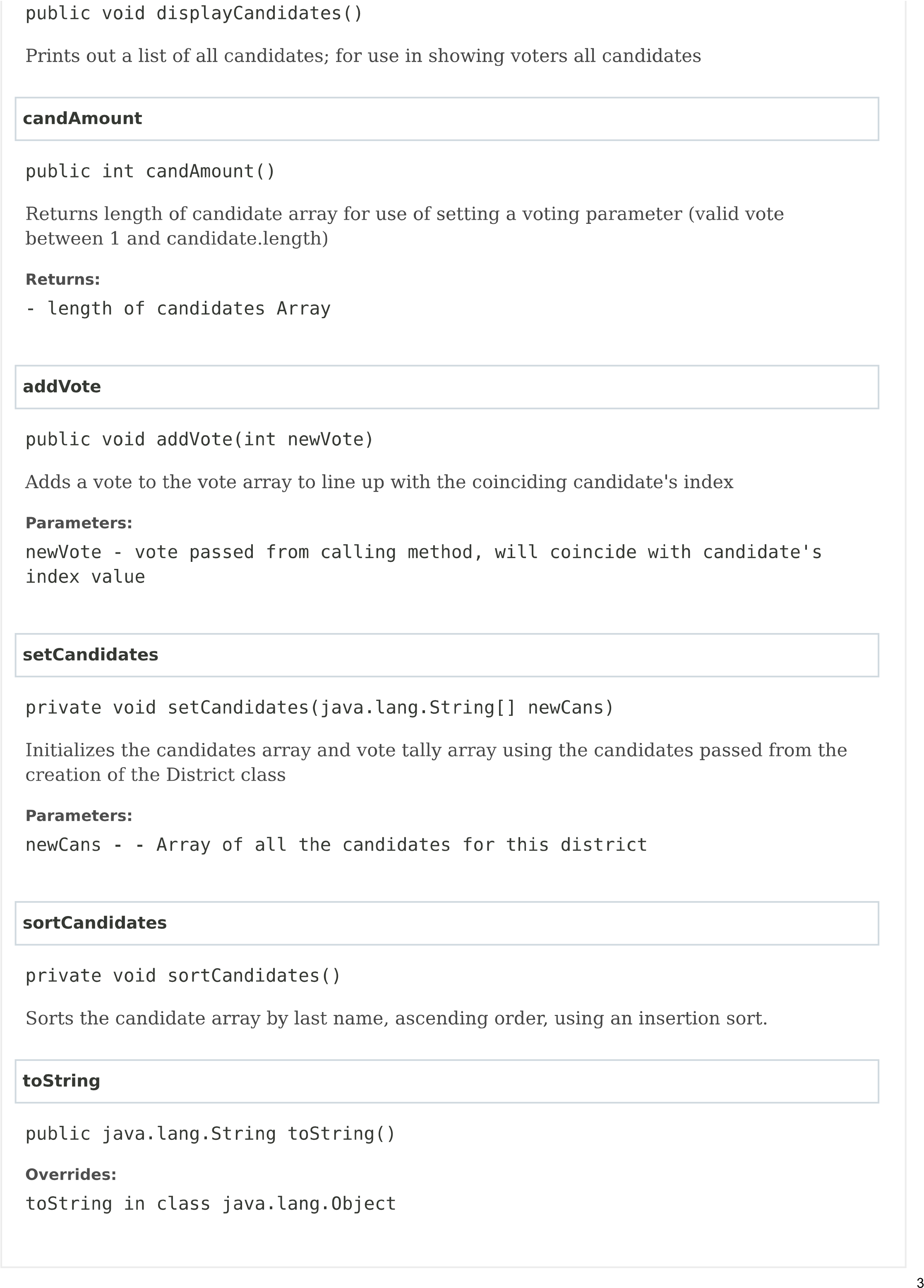
User Manual

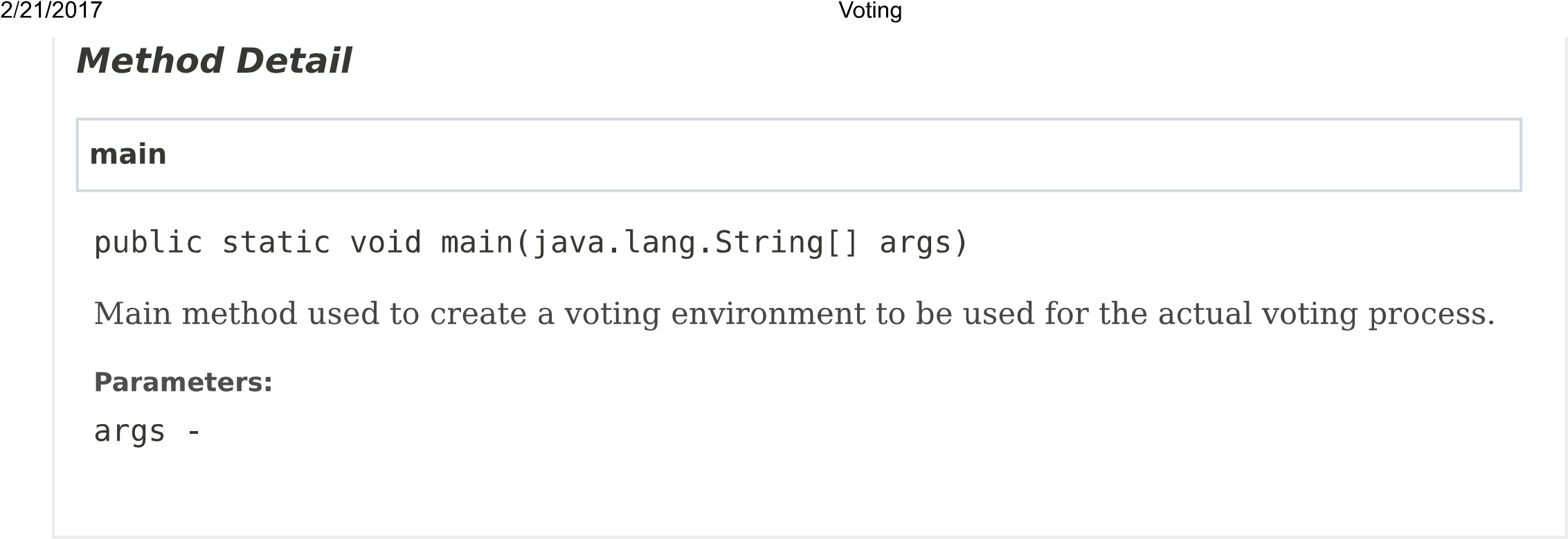
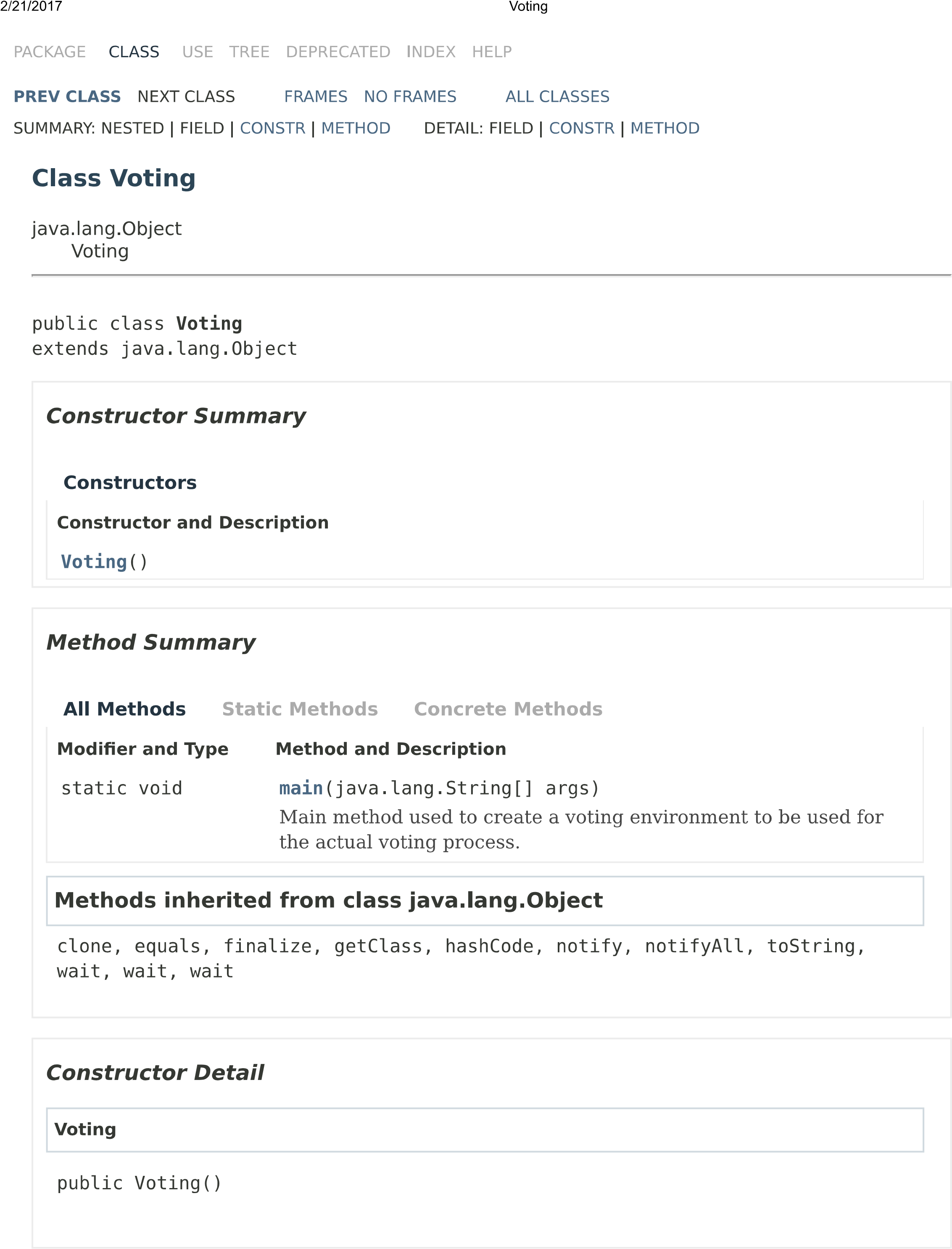
Our working instruction manual would proceed as follows: First, an administrator must input a numeric key (of their choosing) to begin the voting period. The administrator must then input the number of districts, as well as the names of all candidates running for each district. The booth would then be open for voting, and a prompt would appear for the voter’s district. Once the voter inputs the desired district, the program will display the names for all candidates for the given district coinciding with an index number. The voter will then be prompted two choose an index number that corresponds with the candidate they would lie to vote for. In doing so the voter will cast their vote for the chosen candidate, and will be thanked for voting. The program will then display another prompt asking again for (assuming a new voter is occupying the booth) district input. This process will repeat until the administrator inputs a numeric code to stop the voting period.

Javadoc API for both the District and the Voting class on next few pages, for developer use.









Maintenance Step

The final phase of the software life cycle methodology is that of performing maintenance on the developed software. This step includes identifying problems or discrepancies within the finished code, posing solution to these issues, and then implementing them with proper documentation. Since a program can have the potential to be used over a long period, keeping the program up to date is a very important part of the software’s life cycle. It is nearly impossible to foresee every issue that may arise throughout the software’s use, so maintaining it to certain degree becomes absolutely necessary.

Maintenance Document

Due to some of the ambiguous instructions we were giving when creating our voting software, a few issues are apparent. Firstly, what sort of interface did the client want? Our group decided to use a Scanner to take input just for the sake of time, as well as the relative size of the program. However, I think a graphical user interface (GUI) would be much better suited to record votes, due to familiarity that most users would have in using a GUI, as well as the modular approach one could take while designing a GUI. This modular design would be well equipped to handle additional functionality to the code, in the very likely scenario that new functioning will be added to the code. Case in point, my partner and I think that some sort of checklist system should be added to this program. By prompting a voter, most likely a student in this environment, to enter a student ID in the interface, the program could cross check that individual’s student ID to ensure that this student has not voted yet. This would prevent any student from voting twice, and in turn make the software much more reliable. My partner and I also considered the timing of the voting period. We were thinking of implementing a sort of timing method that would read the time from the computer’s internal clock to start and stop the voting period. An administrator would be able to start the voting using a predetermined code, and after a certain elapsed time, the voting would close.