# Assignment 02

This assignment requires you to write a single program in a single Eclipse project directory. Begin by downloading and importing the template code from Canvas into a single project in Eclipse; you should rename the project to `cs220-assign02-<lastName>`, where `<lastName>` is written in camelcase[1] and the first letter is lowercase. For example, if your last name is "Von Neumann", your project should be called `cs220-assign02-vonNeumann`. You must complete both programs and turn them into Canvas as a single compressed archive.

## Preface

Abstract classes and interfaces each have their uses when working with a large class hierarchy; abstract classes help to define the properties and behaviors subclasses should have without needing to provide specifications for those behaviors, while interfaces allow us to define the functionality one or more classes should be able to perform. In this assignment, we are going to build several mock classes for some electronics megacorporation that develops and sells a variety of electronic devices (e.g., smartphones, TVs, tablets). To keep our assignment manageable, we are going to focus on just TVs and smartphones. Our example will also look at incorporating interfaces for both voice assistants and logging (i.e., which devices employ a voice assistant? which devices employ logging?). These components will all be incorporated into a single program.

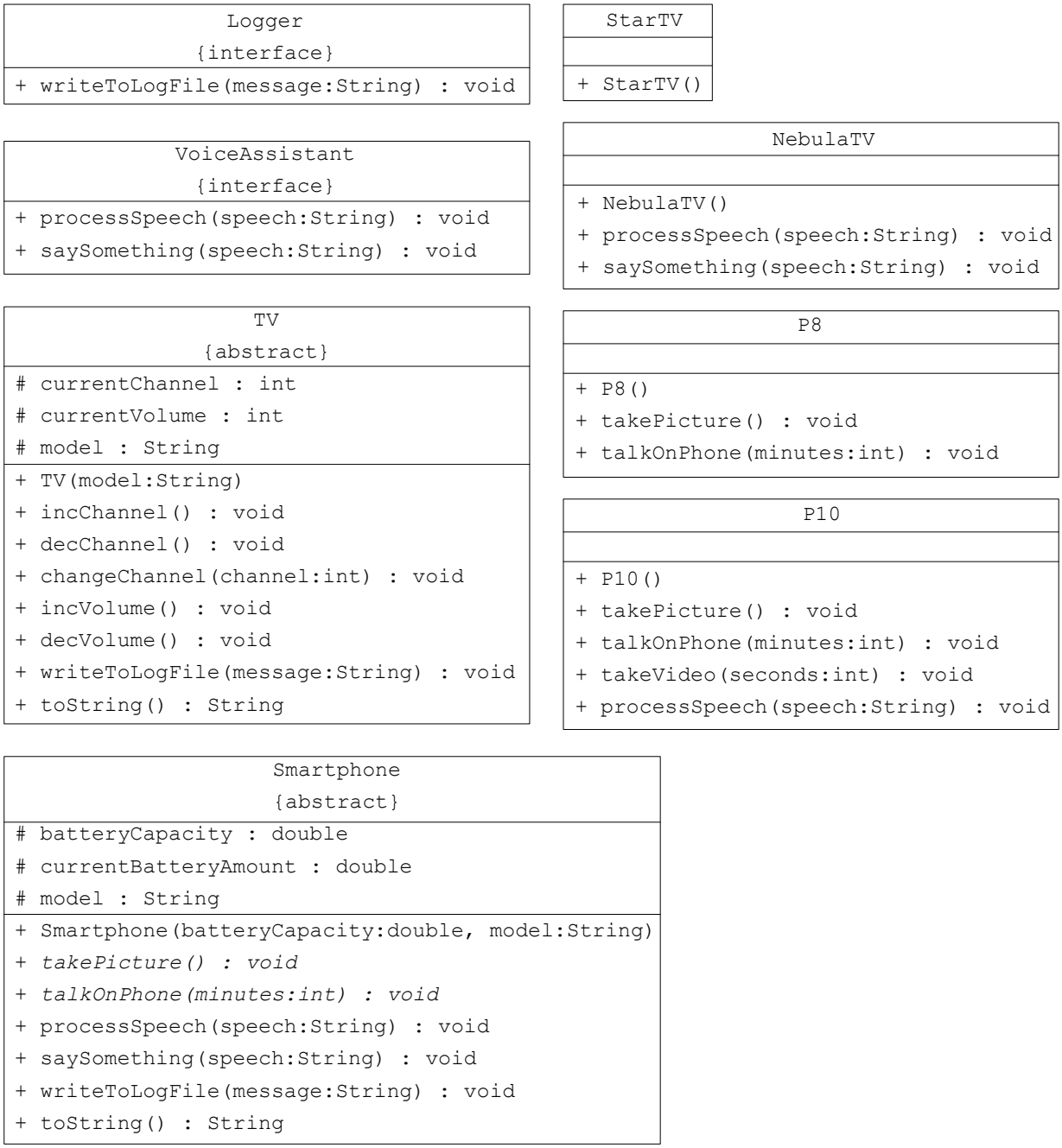
## Program 1 (40 points): `Assign02Test`

The main method for this program will be contained in `Assign02Test`, which should be the only file included in the template downloaded from Canvas. Inside `main`, there is code commented out. This code will ultimately be what runs and tests the classes and interfaces you will write in this assignment. The testing code is split into multiple sections, which you will be asked to comment/uncomment throughout this document to test your code.

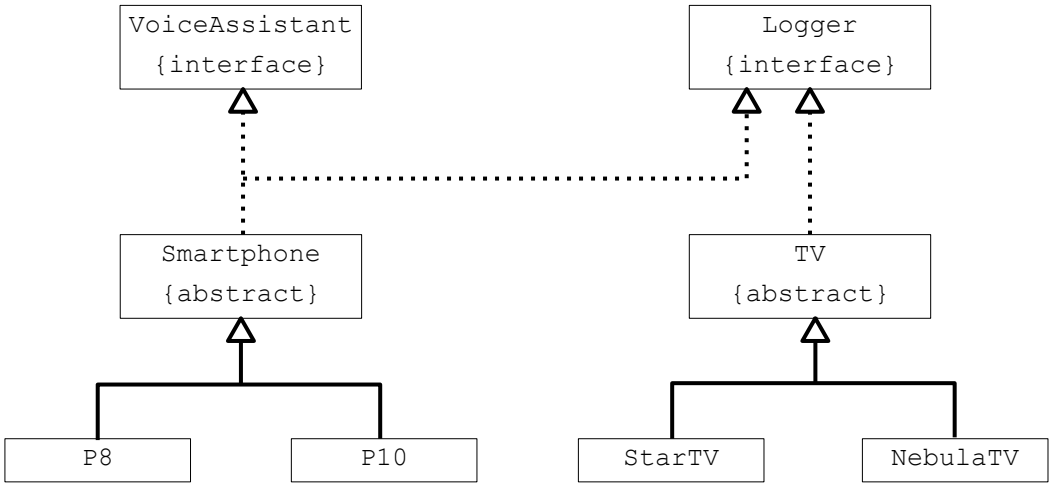
Below is a UML diagram representing the classes, abstract classes, and interfaces and their relationships you will created as part of this assignment. Note that `Assign02.java`, which is provided for you and contains `main`, is not included. You will also write the class `SpeechNotUnderstoodException`, which is also not included. Refer back to these diagrams throughout the assignment to ensure that your classes and method signatures match what is displayed below.

[1] More info on camelcase: https://en.wikipedia.org/wiki/CamelCase

## Class Diagrams



## Inheritance Hierarchy




The assignment is broken down into many steps. While you do not have to complete the steps in this order, they are presented in the order that will make it easiest to move from one part of the program to the next. However, if you get stuck, please see if there are additional steps you can complete while waiting for help.

If you are starting this program substantially early[2] (i.e., while we are still discussing file i/o), note that steps 1-6 and 8, 9, 11, and 12 can mostly be written before we finish discussing file i/o (with the exception of the `writeToFile` method in step 4). The remaining steps and parts of steps can be completed as we finish topics associated with file i/o.

### A Note About This Program

Unlike other programs you have written (and will write this semester), **this program does not have much actual programming** where you need to figure out an algorithm for some problem. Rather, this program is about understanding and building the relationships between classes, abstract classes, and interfaces. The implementations of some of these methods are **very** simple - the intent is to simulate what might happen in the method, and instead focus on the relationships between the classes/interfaces. Please be **very careful** in your reading of this assignment, test as you go, and stop in at office hours if you encounter problems!

### A Note About Commenting

You will be responsible for commenting all classes and methods in this program. This includes methods I direct you to write, as well as any additional (private) methods you may write to help. It also includes abstract methods in abstract classes, and method signatures in interfaces. You may choose to copy and paste a comment from an abstract class/interface into where it is eventually implemented, and embellish with additional details about that implementation. The only methods you do not need to comment are `toString` methods. All class/method comments should follow the guidelines sets forth in the style guide on Canvas.

### Step 1: The `Logger` Interface

Create a new interface named `Logger` in your project. This interface should have a single `void` method called `writeToLogFile(String message)`. Implementations of this method will write the message argument to the appropriate log file. This interface will ensure that some products this company develops have the capability to write to a log file. In our example, that will eventually be

[2]Something I highly encourage!

for all products, but placing this functionality in an interface ensures that we have the option to implement logging capabilities as desired.

### Step 2: The `VoiceAssistant` Interface

Create a new interface named `VoiceAssistant` in your project. This interface will allow the company to selectively add void assistant capabilities to particular products. This interface will have two methods: a `void` method `processSpeech(String speech)` that processes a voice command from a user, and a `void` method `saySomething(String speech)` that will allow the device to say something in reply to the user. To keep our assignment simple, we're going to simulate speech input and output using strings of text.

### Step 3: The `TV` Abstract Class

Create a new abstract class called `TV`. This class should implement the `Logger` interface. `TV` should have at three attributes to track the current channel (`int`), current volume (`int`), and model name (`String`). The constructor for this class should set the current channel to 2, the current volume to 10, and the model should be set to the parameter value passed into the constructor.

### Step 4: Methods for `TV`

The first five methods for `TV` serve to modify the channel and volume. `incChannel()` should increment the channel by 1, `decChannel()` should decrement the channel by 1, `incVolume()` should increment the volume by 1, `decVolume()` should decrement the volume by 1, and `changeChannel(int channel)` should reset the channel to the given argument. Note that the channel and volume values should never fall below 1 (i.e., 1 is the lowest value either channel or volume can take on). Channel and volume cannot go above 50 (i.e., 50 is the highest value either channel or volume can take on).

Implementing the `Logger` interface requires implementation of the method `writeToLogFile(String message)`. All TV units write messages to text log files that are in the same directory as our Java program. The name of the file should be `<modelName>.txt`, where `<modelName>` is taken from the attribute. We will assume that our program will never have more than one instance of a given class (and thus, model of TV). In this method, append to the log file the message argument, then flush and close access to the file. Once this method is implemented, go back to the first five methods and write the action to the log file using this method after incrementing/decrementing/changing the channel/volume, but only if the value changed. The messages written to the log file should be as follows:

```
"increasing channel to 49"  // assuming 49 is the new current channel
"decreasing channel to 50"  // assuming 50 is the new current channel
"cannot increase channel"   // cannot go above 50
"changing channel to 42"    // assuming 42 is the next current channel
"cannot change channel to -5"    // cannot go below 1
"increasing volume to 2"    // assuming that 2 is the new current volume
"decreasing volume to 1"    // assuming that 1 is the next current volume
"cannot decrease volume"    // cannot decrease lower than 1
```

Finally, overriding the `toString()` method will be helpful for me to periodically check the state of your objects. While you are expected to know how to override this method on your own, Eclipse provides a helpful shortcut to do this for you. In Eclipse, go to the "Source" menu, then select "Generate toString()...". Make sure the only attributes (called "fields" here) that are chosen are your variables that represent the current channel, current volume, and model name.

Now that we have the `TV` abstract class set up, we can focus on the two TV models sold by our company: the basic Star TV model, and the Nebula TV model, which has voice assistant capabilities.

### Step 5: The `StarTV` Class

The Star TV is the most basic TV model offered by our company. Create a new class called `StarTV`, which should extend `TV`. No additional attributes are necessary, and the only method should be the constructor, which has no parameters, and should set the model in the parent constructor to `"star"`.

At this point, you should have enough working to use the Star TV tests in `main` (test 1). Output for those tests is near the end of the document.

### Step 6: The `NebulaTV` Class

The Nebula TV is similar to the Star TV, but offers voice assistance. Create a new class called `NebulaTV` which extends `TV` and implements `VoiceAssistant`. No additional attributes are necessary. The constructor should be similar to the constructor for `StarTV`, in that it will have no parameters and should set the model in the parent constructor to `"nebula"`.

Due to implementing the `VoiceAssistant` interface, this class must also provide an implementation for the two methods in the interface. The method `saySomething(String speech)` should print the speech to the console (thus representing something the device would say), and then it should write to the log file `assistant: "<speech>"` (just a single space after the colon), where `<speech>` is the text passed into the method. Note that the log file should include quotes around the speech.

Finally, the method `processSpeech(String speech)` will take in a user's voice command, identify what the command is asking, and then respond by first having the device say what it is doing by using the method `saySomething`, and then performing the associated action. The below table indicates what speech this method should be capable of recognizing, what the device should say in response, and what action the method should then take:

| **input to `processSpeech`** | **device speech and action** |
|---:|---|
| "increase channel" | "increasing channel"/call to `incChannel()` OR "invalid channel, cannot increase" |
| "decrease channel" | "decreasing channel"/call to `decChannel()` OR "invalid channel, cannot decrease" |
| "increase volume" | "increasing volume"/call to `incVolume()` OR "invalid volume, cannot increase" |
| "decrease volume" | "decreasing volume"/call to `decVolume()` OR "invalid volume, cannot decrease" |
| "change <int>" | "changing channel to <int>"/call to `changeChannel(int)` OR "invalid channel, cannot change" |

A few notes about the input cannot change. First, you should check that the user provided a valid value in the change case according to the guidelines outlined in step 4 above. Second, since your channel will be a `String`, you will need to use `Integer.parseInt(<String>)`, which takes a `String` as input and returns the corresponding `int`. You can assume that this last voice command will only ever provide a valid `int` as input, and never anything else.

Finally, this method needs some way of handling completely irrelevant voice commands (i.e., speech input that doesn't fall into one of the five scenarios above). In this case, the device should say

"exception occurred" and throw a new exception. Because the exceptions provided by Java do not adequately fit our needs, we will create a new exception in the next step.

### Step 7: The `SpeechNotUnderstoodException` Class

Create a new class called `SpeechNotUnderstoodException` that extends `Exception`. There should be no attributes, and the only method should be the constructor. The constructor will have a single parameter, `String message`, and will then pass that argument to the parent constructor.

Once you have finished this step, go back to where you were throwing the exception in step 6. The message passed into the exception instantiate to throw should be `"<speech>" is not understood by the voice assistant for <model> tvs.` Notice that successfully adding this will require modifying the method signature for `processSpeech` as well to indicate that this method may throw a checked exception.

At this point, you should have enough working to use the Nebula TV tests in `main` (tests 2 and 3). Output for those tests is near the end of the document.

### Step 8: The `Smartphone` Abstract Class

Having finished the two TV models our company offers, we are now going to shift our attention to the smartphone models offered by our company. Begin by creating a new abstract class called `Smartphone` that implements both `Logger` and `VoiceAssistant`.

Our two smartphone models (the P8 and P10) are going to be very similar in terms of capabilities; both will feature logging and voice assistants. Thus, we concentrate those functionalities in our `Smartphone` abstract class. Additionally, all smartphones will keep track of three attributes: a `String` to keep track of the smartphone model, a `double` to keep track of the battery capacity (i.e., how much charge can the battery hold, measured in mAh), and a `double` to keep track of the current battery amount (i.e., how much charge is currently in the battery, measured in mAh). Go ahead and create these attributes in the `Smartphone` class.

The constructor for the `Smartphone` class will take in values for the total battery capacity and the model. Go ahead and create the constructor method to set the attributes you just created. The current battery capacity should be set to the total battery capacity (i.e., the phone is charged).

The API in `Smartphone` will concentrate on the amount of power usage required for different functionality and recharging the phone. Because `Smartphone` implements `VoiceAssistant`, and those methods will share some similarities amongst child classes, we will implement some of those methods in this class (to be done in later steps). We will also add some additional methods, described below.

First, `Smartphone` will have two abstract methods, `takePicture` which has no parameters, and `talkOnPhone` which has a single `int` parameter indicating the number of minutes talked on the phone. These methods are abstract because the specifics of how the phone takes pictures or uses the phone are dependent on technology in that phone. However, we will not focus on that in our implementation.

Finally, `Smartphone` will also have a single `void` method `charge` which accepts no parameters. This method will first print out the current percentage full that the battery is at, followed by the message "Recharging". Then, it will reset the attribute that tracks the current battery amount to the full battery amount. The printed statements should look like this on the console:

```
The phone battery is at 80.0% // this value should be calculated
Recharging
```

This method should then write ``charging'' to the log file. (we will implement the `writeToLogFile` method for `Smartphone` in a later step)

### Step 9: Additional Methods for `Smartphone`

There are three methods that we need to override from the implemented interfaces: two from `VoiceAssistant` and one from `Logger`. We will focus on the `VoiceAssistant` methods first.

The first method is the `saySomething` method, which takes in a single `String` parameter representing the user's speech. Like in `NebulaTV` class (step 6), this method will just print the parameter's value to the console. The method should then write to the log file `assistant: "<speech>"`, where `<speech>` is the text passed into the method. Note that the log file should include quotes around the speech. (we will implement the `writeToLogFile` method for `Smartphone` later)

The second method is `processSpeech`, which takes in a single `String` speech parameter, representing the user's speech. The user's speech will take one of the following forms in the `Smartphone` class (i.e., these are voice commands recognized by all smartphones), and what the associated action should be (note that the smartphone won't say anything back like the `TV` class does):

| **input to `processSpeech`** | **action** |
|---:|---|
| "picture" | call to `takePicture()` |
| "talk <int>" | call to `talkOnPhone(int)` |

You can assume any voice command `talk` will always have a valid `int`. If the command matches neither of these commands, then you should write to the log file `exception: "<speech>"`, where `<speech>` is the text passed into the method. Note that the log file should include quotes around the speech. (we will implement the `writeToLogFile` method for `Smartphone` later) The method should then throw a new `SpeechNotUnderstoodException` The message passed into the exception instantiate to throw should be `"<speech>" is not understood by the voice assistant for <model> smartphones.` Notice that successfully adding this will require modifying the method signature for `processSpeech` as well to indicate that this method may throw a checked exception.

Finally, override the `toString()` method using the same approach as discussed in step 4.

### Step 10: `writeToLogFile` Method for `Smartphone`

Like the `writeToLogFile` methods for `TV`, this method will output logging messages (passed in as a `String` parameter) to a file names with the model of the smartphone. (we will never have more than one smartphone of a particular model) In this case, we will write out the files as binary files, since smartphones have space constraints, and binary files will take up less room. Thus, these files will be named `<modelName>.bin`, where `<modelName>` is taken from the attribute. Be sure to always append to these files.

When writing the binary output for the method, you should consider the two possible types of values you will have - `String` text, and `int` numbers. Your method will need to differentiate between the two in order to determine whether to use `writeInt` or `writeChars`. A `private` method you might include in your program to help with this task is included below:

```
private static boolean isInt(String s) {
    try {
        Integer.parseInt(s);
    } catch(NumberFormatException e) {
        return false;
    }
    return true;
}
```

You might find this method easier to write after finishing steps 11 and 12 so you can see the types of messages being written to the log file.

### Step 11: The `P8` Class

The P8 smartphone is the basic smartphone offered by our company. This class should extend `Smartphone`. The constructor method should have no parameters, and should pass in the value `2000` for the battery capacity and `"p8"` for the model.

The method `takePicture` should provide an implementation for the abstract method in the `Smartphone` class. This method should first print to the console `<model>: Taking picture at 1200 x 900 pixels.` The method should then subtract `200` from the current battery amount. Finally, it should write to the log file `"picture"`.

The method `talkOnPhone` should provide an implementation for the abstract method in the `Smartphone` class. This method should first print to the console `<model>: Talking on phone for <minutes> minutes.` The method should then subtract `20 * minutes` from the current battery amount. Finally, it should write to the log file `"talk: <minutes>"`.

At this point, you should have enough working to use the P8 tests in `main` (tests 4 and 5). Output for those tests is near the end of the document.

### Step 12: The `P10` Class

The P10 smartphone is the more advanced smartphone offered by our company. It has improved battery life and efficiency over the P10 and can also take videos. Start by extending the `Smartphone` class. Implement the constructor like you did for the P8 smartphone (i.e., with no parameters), but this time using the value `2900` for the battery capacity and `"p10"` for the model.

The method `takePicture` should provide an implementation for the abstract method in the `Smartphone` class. This method should first print to the console `<model>: Taking picture at 1500 x 1125 pixels.` The method should then subtract `150` from the current battery amount. Finally, it should write to the log file `"picture"`.

The method `talkOnPhone` should provide an implementation for the abstract method in the `Smartphone` class. This method should first print to the console `<model>: Talking on phone for <minutes> minutes.` The method should then subtract `10 * minutes` from the current battery amount. Finally, it should write to the log file `"talk: <minutes>"`.

The P10 will also have a new method, `takeVideo` that has a single `int` parameter, the length of the video in seconds, and does not return anything. This method should first print to the console `<model>: Taking video on phone for <seconds> seconds` The method should then subtract `5 * seconds` from the current battery amount. Finally, it should write to the log file `"video: <seconds>"`.

Because the P10 smartphone also takes video, we should override the `processSpeech` method to also deal with the voice command for recording video. The user's speech will take the following form for recording video on the P10:

| **input to** `processSpeech` | **action** |
|---|---|
| "video <int>" | call to `takeVideo(int)` |

Notice that this phone should **also** recognize the commands associated with `Smartphone`. Your method should first see if the command is to record video. If it is, deal with it as described in the above table. If it is not, this method should call the `processSpeech` in the parent class. Note that this method also has the potential to throw the `SpeechNotUnderstoodException`.

At this point, you should have enough working to use the P8 tests in `main` (tests 6 and 7). Output for those tests is near the end of the document.

### Testing Type Conformance

At this point, you have implemented all the relationships and functionality for this program. The remaining tests in `main` (tests 8, 9, 10, 11) each test for type conformance of the different classes with the abstract classes and interfaces. If you uncomment one of these tests and have syntactic errors, then there is an error in either one of your class relationships (e.g., missing an `extends`/`implements`), or in your overriding of a method. Please revisit the pertinent document sections. Output for these tests is at the end of this document.

## Program Output

Below is the result of the console after each of the tests - I highly recommend only uncommenting one test at a time when working on this program. I also show the results of the log files for tests 1-7, which test the individual classes. Logging should continue to work and follow for tests 8-11. **You will need to delete these log files each time you run the program/tests in order to match my output**. This can be done through Eclipse or on your file system. Also note that, due to space constraints, some lines are wrapped in the example output below, but should not be wrapped in your output (wrapped lines are indented).

**Test 1**
**Console Output**

```
** TEST 1 **
TV [currentChannel=1, currentVolume=1, model=star]
```

star.txt

```
decreasing channel to 1
increasing channel to 2
changing channel to 50
cannot increase channel
changing channel to 1
cannot decrease channel
increasing volume to 11
decreasing volume to 10
decreasing volume to 9
decreasing volume to 8
decreasing volume to 7
decreasing volume to 6
decreasing volume to 5
decreasing volume to 4
decreasing volume to 3
decreasing volume to 2
decreasing volume to 1
cannot decrease volume
```

**Test 2**
**Console Output**

```
** TEST 2 **
TV [currentChannel=1, currentVolume=10, model=nebula]
```

nebula.txt

```
decreasing channel to 1
increasing channel to 2
changing channel to 50
cannot increase channel
changing channel to 1
cannot decrease channel
increasing volume to 11
decreasing volume to 10
```

**Test 3**
**Console Output**

```
** TEST 3 **
increasing channel
decreasing channel
increasing volume
decreasing volume
changing channel to 50
invalid channel, cannot increase
exception occurred
"hello tv" is not understood by the voice assistant for nebula tvs.
TV [currentChannel=50, currentVolume=10, model=nebula]
```

nebula.txt

```
assistant: "increasing channel"
increasing channel to 3
assistant: "decreasing channel"
decreasing channel to 2
assistant: "increasing volume"
increasing volume to 11
assistant: "decreasing volume"
decreasing volume to 10
assistant: "changing channel to 50"
changing channel to 50
assistant: "invalid channel, cannot increase"
assistant: "exception occurred"
exception: hello tv
```

**Test 4**
**Console Output**

```
** TEST 4 **
p8: Taking picture at 1200 x 900 pixels.
p8: Talking on phone for 10 minutes.
The phone battery is at 80.0%
Recharging
Smartphone [batteryCapacity=2000.0, currentBatteryAmount=2000.0, model=p8]
```

p8.bin **(note that some characters you would expect to see will not render in a text editor)**

```
picture
talk:

charging
```

**Test 5**
**Console Output**

```
** TEST 5 **
p8: Taking picture at 1200 x 900 pixels.
p8: Talking on phone for 5 minutes.
"video 30" is not understood by the voice assistant for p8 smartphones.
Smartphone [batteryCapacity=2000.0, currentBatteryAmount=1700.0, model=p8]
```

`p8.bin` **(note that some characters you would expect to see will not render in a text editor)**

```
picture
talk:
exception: video
```

**Test 6**
**Console Output**

```
** TEST 6 **
p10: Taking picture at 1500 x 1125 pixels.
p10: Talking on phone for 10 minutes.
p10: Taking video on phone for 15 seconds.
The phone battery is at 88.0%
Recharging
Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2900.0, model=p10]
```

`p10.bin` **(note that some characters you would expect to see will not render in a text editor)**

```
picture
talk:

video:
charging
```

**Test 7**
**Console Output**

```
** TEST 7 **
p10: Taking picture at 1500 x 1125 pixels.
p10: Talking on phone for 5 minutes.
p10: Taking video on phone for 30 seconds.
"camera 15" is not understood by the voice assistant for p10 smartphones.
Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2550.0, model=p10]
```

`p10.bin` **(note that some characters you would expect to see will not render in a text editor)**

```
picture
talk:
video:
exception: camera
```

**Test 8**
**Console Output**

```
** TEST 8 **
TV [currentChannel=2, currentVolume=10, model=star]
TV [currentChannel=2, currentVolume=10, model=nebula]
TV [currentChannel=13, currentVolume=10, model=star]
TV [currentChannel=13, currentVolume=10, model=nebula]
```

**Test 9**
**Console Output**

```
** TEST 9 **
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=2000.0,
    model=p8]
p8: Talking on phone for 10 minutes.
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=1800.0,
    model=p8]
The phone battery is at 90.0%
Recharging
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2900.0,
    model=p10]
p10: Talking on phone for 10 minutes.
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2800.0,
    model=p10]
The phone battery is at 96.0%
Recharging
```

**Test 10**
**Console Output**

```
** TEST 10 **
This is the TV [currentChannel=2, currentVolume=10, model=nebula]
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=2000.0,
    model=p8]
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2900.0,
    model=p10]
```

**Test 11**
**Console Output (yes, this is correct)**

```
** TEST 11 **
```

**All Commands in `main` Uncommented (i.e., all tests)**

```
** TEST 1 **
TV [currentChannel=1, currentVolume=1, model=star]


** TEST 2 **
TV [currentChannel=1, currentVolume=10, model=nebula]


** TEST 3 **
increasing channel
```

```
decreasing channel
increasing volume
decreasing volume
changing channel to 50
invalid channel, cannot increase
exception occurred
"hello tv" is not understood by the voice assistant for nebula tvs.
TV [currentChannel=50, currentVolume=10, model=nebula]


** TEST 4 **
p8: Taking picture at 1200 x 900 pixels.
p8: Talking on phone for 10 minutes.
The phone battery is at 80.0%
Recharging
Smartphone [batteryCapacity=2000.0, currentBatteryAmount=2000.0, model=p8]


** TEST 5 **
p8: Taking picture at 1200 x 900 pixels.
p8: Talking on phone for 5 minutes.
"video 30" is not understood by the voice assistant for p8 smartphones.
Smartphone [batteryCapacity=2000.0, currentBatteryAmount=1700.0, model=p8]


** TEST 6 **
p10: Taking picture at 1500 x 1125 pixels.
p10: Talking on phone for 10 minutes.
p10: Taking video on phone for 15 seconds.
The phone battery is at 88.0%
Recharging
Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2900.0, model=p10]


** TEST 7 **
p10: Taking picture at 1500 x 1125 pixels.
p10: Talking on phone for 5 minutes.
p10: Taking video on phone for 30 seconds.
"camera 15" is not understood by the voice assistant for p10 smartphones.
Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2550.0, model=p10]


** TEST 8 **
TV [currentChannel=1, currentVolume=1, model=star]
TV [currentChannel=50, currentVolume=10, model=nebula]
TV [currentChannel=13, currentVolume=1, model=star]
TV [currentChannel=13, currentVolume=10, model=nebula]



** TEST 9 **
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=1700.0,
    model=p8]
p8: Talking on phone for 10 minutes.
```

```
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=1500.0,
    model=p8]
The phone battery is at 75.0%
Recharging
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2550.0,
    model=p10]
p10: Talking on phone for 10 minutes.
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2450.0,
    model=p10]
The phone battery is at 84.0%
Recharging



** TEST 10 **
This is the TV [currentChannel=13, currentVolume=10, model=nebula]
This is the Smartphone [batteryCapacity=2000.0, currentBatteryAmount=2000.0,
    model=p8]
This is the Smartphone [batteryCapacity=2900.0, currentBatteryAmount=2900.0,
    model=p10]



** TEST 11 **
```

## Grading:

Your assignment will be graded on:

- The degree to which your program meets the functional requirements.
- Inclusion of appropriate comments (class/method/header and inline comments), and formatting according to the style guide on Canvas.
- **Code with syntax error (i.e., red underlines) will receive a grade of 0 for the functional requirements.** It is near impossible to grade code that does not run; be sure to leave ample time for office hours if you struggle with writing syntactically correct code.
- Exceptions will incur a 5% deduction from your final grade.

## Submission

You will submit a **single Eclipse archived project** containing all of the source code for all of the programs in this assignment in a zip compressed format (i.e., .zip) to Canvas. The zip should be named **exactly** `cs220-assign02-<lastName>`, where `<lastName>` is written in camelcase[3] and the first letter is lowercase. For example, if your last name is "Von Neumann", your submission will be named `cs220-assign02-vonNeumann`. **A 5% penalty** will be applied if your submission is in a different form than an archived project file.

[3] More info on camelcase: https://en.wikipedia.org/wiki/CamelCase