PA 8 - Generics and Exceptions

Assignment Overview

Due date: Thursday, March 5th @ 11:59PM

This assignment is intended to further your understanding of using Java generics and exception that are commonly seen in data structure implementations. It should give you a sense of different ways to make your classes flexible to different data types and other ways to handle exceptions compared to traditional if/else approach.

Getting Started

There are a number of ways to get started on development. The following is the recommended way to ensure that your code will compile during grading.

1. If you are using your own machine or are on a lab computer to complete the assignment, go to step 2 directly. Otherwise, ssh into your cs8bwi20 account.
    - `ssh cs8bwi20__@ieng6.ucsd.edu`
2. Acquire the starter files.
    - From ieng6:
        - Log in to your cs8bwi20 account.
        - From the command line, use the command `cp -r ~/../public/pa8 ~/` (this will copy the entire starter files directory to your home directory)
        - Type `ls ~` to verify that you have copied the `pa8` directory over.
    - From GitHub:
        - `git clone https://github.com/CaoAssignments/cse8b-wi20-pa8-starter.git`
        - Alternatively, you can download the repo as a zipped folder.
3. If you downloaded the repo as a zipped folder, navigate to it through your terminal or text editor (Atom, Eclipse, etc.). If you git cloned the repo, you can switch into that directory immediately.
    - `cd cse8b-wi20-pa8-starter`
    - Optional: You may choose to rename this repo. You can do this by using this command (before the `cd` command above):
        - `mv cse8b-wi20-pa8-starter pa8`
4. You can now start working on it through vim using the following command or open the directory in your preferred editor.
    - `vim Pokemon.java` or `gvim Pokemon.java`
5. To compile your code, use the `javac` command.
    - Syntax: `javac file1.java file2.java etc...`
    - Example: `javac Simulator.java`
6. To run your code, use the `java` command passing in the name of the class with the main method that you want to run.
    - Syntax: `java nameOfClass`
    - Example: `java Simulator`

Concepts

Generics

A generic type is a generic class or interface that is parameterized over types. By convention, type parameter names are single, uppercase letters. The most commonly used type parameter names are:

- E - Element (used extensively by the Java Collections Framework)
- K - Key
- N - Number
- T - Type
- V - Value

Generics in Java allow us to write programs that don't confine to a specific data type. By using the generic type <E>, we can use E as if it is a type such as String or Integer. Note the following when using the generic types

1. You can't create objects using the generic type (e.g. new E() isn't allowed). But you can create references using generic type (e.g. E ref; is fine)
2. The generic type can't be part of an instanceof check.

Exceptions

In this PA, you will be required to handle different exceptions using try-catch blocks in your methods. An exception is characterized as anything that disrupts the normal flow of the method. Some examples of different exceptions include NullPointerException, ArithmeticException, and IndexOutOfBoundsException.

In the following code,

```
public class Main {
  public static void main(String[] args) {
    System.out.println(1/0);
  }
}
```

running it will result in message similar like this:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at ...
```

Sometimes if an error occurs, you do not want the program to terminate and instead want to handle the error gracefully and continue the rest of the execution. To do this, you should enclose the part of your code that might cause the exception in your try block. In your catch block, make sure you are catching the appropriate exception. Then, depending on how you want to handle the exception, you can print the exception, throw the exception, etc.

Let's take a method attempting to divide by 0 as an example. To handle the exception in the method, you would do the following:

```
public static void divideByZero() throws ArithmeticException {
  try {
   System.out.println (39 / 0);
  }
  catch (ArithmeticException exception) {
   System.out.println ("Exception caught!");
   throw exception;
  }
}
```

Provided Files

There is one provided file, `Simulator.java`. However, it is incomplete and you are responsible for filling in the rest of the code as described below and described in the file as TODOs.

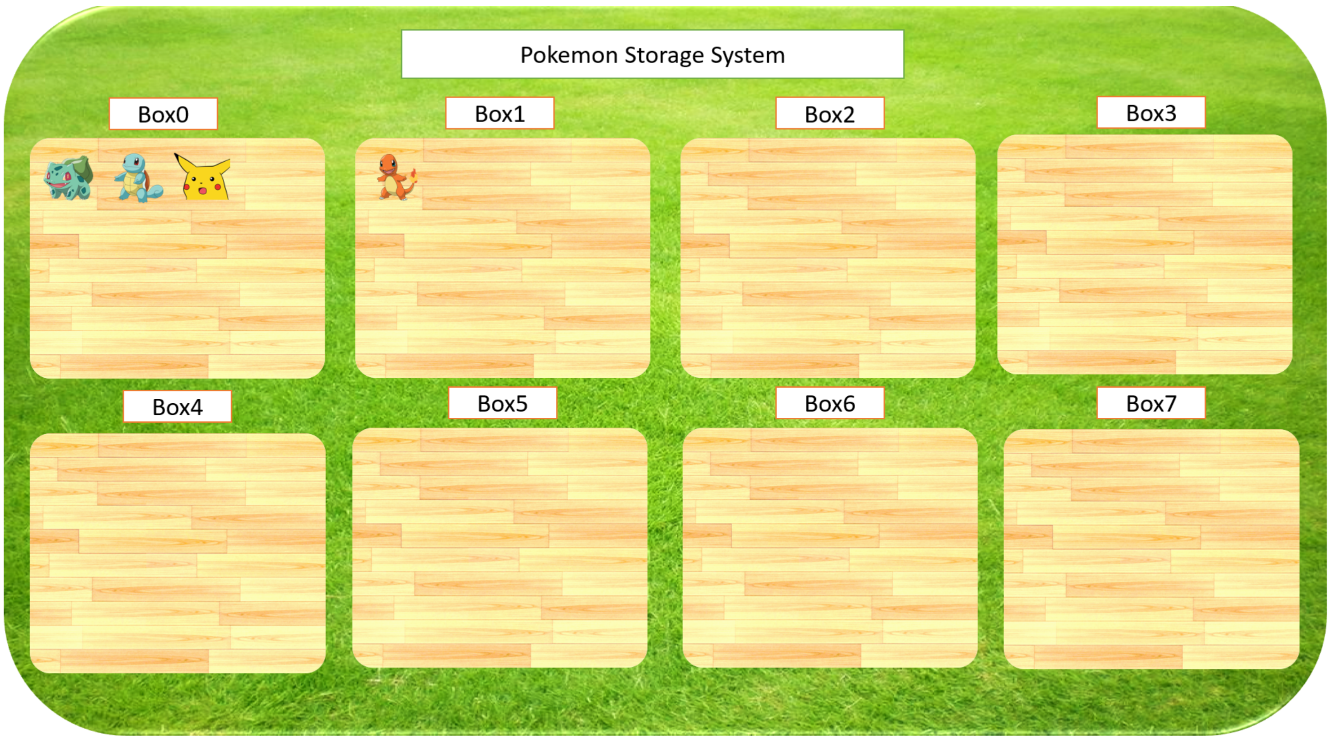
You are responsible for testing all cases in order to ensure that your code works as intended. If you have any questions about the behavior of certain cases that are not addressed in the writeup, please make a Piazza post.

Your Task

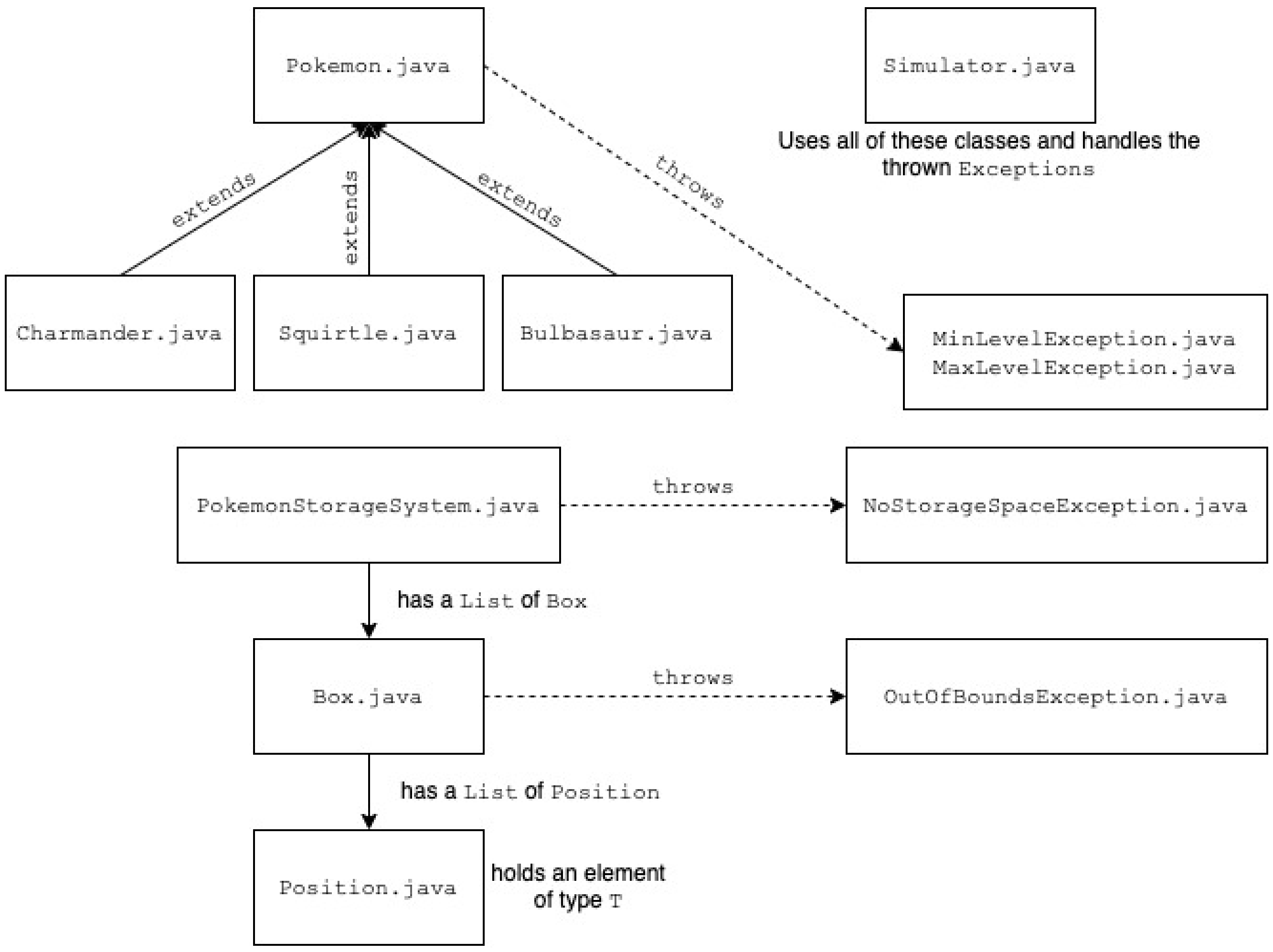
In this PA, you will be completing the implementation of a simplified Pokemon game simulator. In the simulator, you will be able to battle a wild Pokemon, catch a wild Pokemon, and interact with the Pokemon storage system, which trainers can use to store their Pokemons.

Each Pokemon storage system contains a max number of boxes, and each box has a max number of Pokemons it can hold. You can perform a number of operations in the box, such as depositing a Pokemon or releasing one.

Here is a diagram showing the how the Pokemon are stored in boxes in the storage system.



Overall workflow of the PA:



Custom Exception Classes

We will create four of our own exception classes for this assignment. Their names are `MaxLevelException.java`, `MinLevelException.java`, `NoStorageSpaceException.java`, and `OutOfBoundsException.java`. Each class will extend from the `Exception` class and will be quite similar to each other.

MaxLevelException

This exception is used whenever a Pokemon has reached the max level (100) and cannot level up anymore.

Declare the following constant and instance variable in the class:

```
private static final String EXCEPT_MSG = "%s can't be greater than level 100!\n";
private String pokemonName;
```

```
public MaxLevelException(String name)
```

Write a constructor that takes in a single String as a parameter. This String parameter is the name of the Pokemon that caused the MaxLevelException. In the constructor, call the `super()` constructor and pass in a formatted String using `EXCEPT_MSG` and `name`. Then, set your instance variable `pokemonName` to the String that was passed in to this constructor.

```
@Override
public String toString()
```

We will override the `toString()` method so that we can print out the message associated with this exception later. Simply return a formatted String using `EXCEPT_MSG` and `pokemonName`.

Hint: To format a String, `String.format()` is a useful method.

MinLevelException

This exception is used whenever a Pokemon's level is below the minimum level of 1. This class should be extremely similar to MaxLevelException.

Declare the following constant and instance variable in the class:

```
private static final String EXCEPT_MSG = "%s can't be less than level 1!\n";
private String pokemonName;
```

```
public MinLevelException(String name)
```

Write a constructor that takes in a single String as a parameter. This String parameter is the name of the Pokemon that caused the MinLevelException. In the constructor, call the `super()` constructor and pass in a formatted String using `EXCEPT_MSG` and `name`. Then, set your instance variable `pokemonName` to the String that was passed in to this constructor.

```
@Override
public String toString()
```

We will override the `toString()` method so that we can print out the message associated with this exception later. Simply return a formatted String using `EXCEPT_MSG` and `pokemonName`.

NoStorageSpaceException

`NoStorageSpaceException` is used when our Pokemon storage system is complete out of space and cannot store any more Pokemon.

Declare the following constant in the class:

```
private static final String EXCEPT_MSG = "No storage left\n";
```

```
public NoStorageSpaceException()
```

Unlike the previous two exception classes, the constructor of `NoStorageSpaceException` does not take a String parameter. In the constructor, simply call `super()` and pass in `EXCEPT_MSG`.

```
@Override
public String toString()
```

As before, we will override the `toString()` method. Return `EXCEPT_MSG`.

OutOfBoundsException

This exception will be used when the user attempts to access an invalid location in our Pokemon storage system.

Declare the following constants and instance variable in the class:

```
private static final String EXCEPT_MSG = "Out of bounds: %s\n";
private String errorLocation;
```

```
public OutOfBoundsException(String loc)
```

The constructor of `OutOfBoundsException` takes a single String as a parameter, which is the location that is out of bounds. In the constructor, call the `super()` constructor and pass in a formatted String using `EXCEPT_MSG` and `loc`. Then, set your instance variable `errorLocation` to the String that was passed in to this constructor.

```
@Override
public String toString()
```

We will override the `toString()` method so that we can print out the message associated with this exception later. Simply return a formatted String using `EXCEPT_MSG` and `errorLocation`.

Pokemon Classes

Pokemon.java

Pokemon is a concrete class and represents a single Pokemon.

Constants

This class will have two constants:

```
private static final int MAX_DAMAGE = 10;
private static final int MAX_LEVEL  = 100;
```

- MAX_DAMAGE is the maximum damage a Pokemon can inflict.
- MAX_LEVEL is the maximum level a Pokemon can have. It is inclusive, meaning that a Pokemon can be level 100 but not level 101.

Instance Variables

```
private String dexNumber;
private String name;
private int level;
private Random random;
```

- dexNumber is the Pokedex number of the Pokemon. Each Pokemon species has its own unique dexNumber. If you have never played the Pokemon series, you can think of as an id for each Pokemon.
- name is the name or species of the Pokemon. For example, if the Pokemon was a Squirtle, name would be "Squirtle".
- level is the level of the Pokemon. The level of a Pokemon can be between [1, 100], meaning that the lowest the level can be is 1 and the highest the level can be is 100.
- random is a Random object. It will help you decide how much damage your Pokemon will inflict when battling.

Constructor and Methods

```
public Pokemon(String dexNumber, String name, int level)
    throws MinLevelException, MaxLevelException
```

This is the sole constructor of Pokemon. In the constructor:

- Set the dexNumber and name instance variables appropriately.
- For level, we need to check if the level is valid. If the level is less than 1, throw a MinLevelException and pass in name as the argument. If the level is greater than 100, throw a MaxLevelException and pass in name as the argument. Otherwise, the level is valid, so set the instance variable level appropriately.
- Initialize random to a new Random object.

```
public String getName()
```

Return the `name` of this Pokemon.

```
public int getLevel()
```

Return the `level` of this Pokemon.

```
public Random getRandom()
```

Return the `random` object of this `Pokemon`.

```
@Override
public String toString()
```

We will override the `toString()` method. In this method, return the `dexNumber` of this Pokemon.

```
public int attack()
```

This method returns the damage value that this Pokemon will inflict while battling. The damage value will change each time, based on a random number generator that uses `random`. Generate and return a new integer between 0 (inclusive) and `MAX_DAMAGE` (exclusive).

Bulbasaur.java

You are being asked to implement concrete classes for the starter Pokemon of Generation 1. There are three Pokemon in total. Bulbasaur is the grass type starter that you can choose!

`Bulbasaur` is a concrete class that extends `Pokemon`.

Constants

Include the following constants in your file:

```
private static final String NAME       = "Bulbasaur";
private static final String DEX_NUMBER = "001";
private static final int INITIAL_LEVEL = 5;
```

Constructor and Methods

```
public Bulbasaur() throws MinLevelException, MaxLevelException
```

Call the `super()` constructor of Bulbasaur and pass in the appropriate arguments to create a Bulbasaur with starter stats.

```
@Override
public int attack()
```

Override the inherited `attack()` method. The damage value will be randomly decided each time. The damage value will be either 0, 6, or 10. Each value has an equal probability of occuring.

Hint: To generate the random damage value, use `getRandom()` to access the `Random` object from the superclass.

Charmander.java

Charmander is the fire type starter that you can choose!

`Charmander` is a concrete class that extends `Pokemon`.

Constants

Include the following constants in your file:

```
private static final String NAME       = "Charmander";
private static final String DEX_NUMBER = "004";
private static final int INITIAL_LEVEL = 5;
```

Constructor and Methods

```
public Charmander() throws MinLevelException, MaxLevelException
```

Similar to Bulbasaur's constructor, call the `super()` constructor of Charmander and pass in the appropriate arguments to create a Charmander with starter stats.

```
@Override
public int attack()
```

A charmander will always attack with a damage value of 5. Be sure to declare a constant for the damage value to avoid magic numbers.

Squirtle.java

Squirtle is the water type starter that you can choose!

`Squirtle` is a concrete class that extends `Pokemon`.

Constants

```
private static final String NAME       = "Squirtle";
private static final String DEX_NUMBER = "007";
private static final int INITIAL_LEVEL = 5;
```

Constructor and Methods

```
public Squirtle() throws MinLevelException, MaxLevelException
```

Similar to Bulbasaur's and Charmander's constructors, call the `super()` constructor of Squirtle and pass in the appropriate arguments to create a Squirtle with starter stats.

```
@Override
public int attack()
```

A Squirtle will attack with a damage value of either 3 or 8. Each damage value is equally likely, meaning that each damage value occurs with equal probability.

Storage Classes

Position.java

`Position` is a generic class. Make sure it can take in another type by making it have a parameterized type.

Instance Variables

```
private T pokemon;
```

This instance variable represents what Pokemon is currently at this `Position` in the `Box`. (For more information about `Box`, continue reading.) If `pokemon` is null, there is currently not a Pokemon at this `Position`.

Constructor and Methods

```
public Position(T pokemon)
```

Inside of the sole `Position` constructor, set your instance variable to the parameter.

```
public T getPokemon()
```

Return the Pokemon that is at this `Position`.

```
public void setPokemon(T newPokemon)
```

Set the `pokemon` instance variable to `newPokemon` so that `newPokemon` is now the Pokemon at this `Position`.

```
public boolean isOpen()
```

Return `true` if this `Position` is open and `false` otherwise. Recall that we represent an open `Position` if the instance variable `pokemon` is `null`.

Box.java

Similar to `Position`, `Box` is also a generic class. Make sure it can take in another type by making it have a parameterized type.

Constants

`Box` contains constants used in its `toString()` method and a constant used as an argument to an exception. Include the following constants in your file:

```
private static final String BORDER     = "---------------------";
private static final String DIVIDER    = "|";
private static final String NEW_LINE   = "\n";
private static final String EMPTY_POS  = "   ";
private static final int MAX_ELEM_LINE = 5;

private static final String OUT_OF_BOUNDS_EXCEPTION = "Index: %s";
```

Instance Variables

We will create two instance variables that define the functionality of the box that stores Pokemon.

```
private List<Position<T>> boxElements;
private int maxSize;
```

- `boxElements` is a Java List interface type that holds `Position` objects. Note that `Position` is also a generic class that can hold any type.
- `maxSize` keeps track of the maximum number of Pokemon allowed in the box.

Constructor and Methods

```
public Box(int maxSize)
```

This is the sole constructor for the Box class. To implement it, use the parameters passed in to initialize your instance variables. In this constructor, you should:

1. Set the `maxSize` of the box to the value of the parameter.
2. Set `boxElements` to an ArrayList with `maxSize` number of `Position` objects. When creating each `Position` object, pass in `null` as the parameter to the `Position` constructor.

```
@Override
public String toString() {
    int counter = 0;

    StringBuilder boxPrintout = new StringBuilder();
    boxPrintout.append(BORDER);

    // Iterate through each element, print 5 at most on a line
    for(Position<T> element : boxElements) {
        if(counter == 0) {
            boxPrintout.append(NEW_LINE);
            boxPrintout.append(DIVIDER);
        }

        // Print EMPTY_POS if the spot is free (null)
        T pokemon = element.getPokemon();
        if(element.isOpen()) {
            boxPrintout.append(EMPTY_POS);
        } else {
            boxPrintout.append(pokemon.toString());
        }
        boxPrintout.append(DIVIDER);

        counter++;

        // Used so we only have 5 elements at most on a line
        if(counter == MAX_ELEM_LINE) {
            boxPrintout.append(NEW_LINE);
            boxPrintout.append(BORDER);
            counter = 0;
        }
    }
    boxPrintout.append(NEW_LINE);

    return boxPrintout.toString();
}
```

`toString()` is provided to you. Copy the code above and make sure to change the styling of our code snippet to ensure it conforms to the style guideline and your styling.

```
public boolean deposit(T newPokemon)
```

This method will attempt to deposit a Pokemon into the box. Iterate through `boxElements`, and check each `Position` to see if it is open. At the first open position you encounter, set the Pokemon at this position to `newPokemon` and return `true` to indicate success. Otherwise, if there are no open `Position`s in the box, return `false`.

```
public Position<T> getPositionAtIndex(int index) throws OutOfBoundsException
```

First, check if `index` is invalid. `index` is invalid if it is less than zero or greater than or equal to `maxSize`. If `index` is invalid, throw an `OutOfBoundsException` and pass in the String constant `OUT_OF_BOUNDS_EXCEPTION` with the `index`.

If `index` is valid, return the `Position` at `index`.

Hint: To pass in the String message to the exception, use `String.format()` and pass in `OUT_OF_BOUNDS_EXCEPTION` and `index` as its argument to this method.

PokemonStorageSystem.java

Similar to `Box` and `Position`, `PokemonStorageSystem` is a generic class. Make sure it can take in another type by making it have a parameterized type. A `PokemonStorageSystem` consists of a number of `Box`, and each `Box` contains a certain number of `Position`. We also have one `partyMember` Pokemon, which is our currently "active" Pokemon and is used when we battle.

Constants

```
private static final int MAX_BOXES = 8;
private static final int MAX_ITEMS = 30;

private static final String OUT_OF_BOUNDS_EXCEPTION = "Box: %d, Pos: %d";
```

- `MAX_BOXES` is the maximum number of boxes that this storage system can have.
- `MAX_ITEMS` is the maximum number of items that each `Box` can have.
- `OUT_OF_BOUNDS_EXCEPTION` is used as an argument to the `OutOfBoundsException` when necessary.

Instance Variables

```
private List<Box<T>> storage;
private T partyMember;
```

storage is a List of Box and represents our Pokemon storage system. We have the ability to have one Pokemon in our party, and this Pokemon is represented by partyMember.

Constructor and Methods

```
public PokemonStorageSystem()
```

Set storage to a new ArrayList. Add MAX_BOXES number of Box to the ArrayList. For each Box, be sure to pass MAX_ITEMS into the Box constructor.

```
public void setPartyMember(T partyMember)
```

Set the partyMember instance variable to the parameter.

```
public void deposit(T newPokemon) throws NoStorageSpaceException
```

This method will iterate through storage and attempt to deposit newPokemon in a Box. If the Pokemon was successfully deposited in a Box, return. Otherwise, if the Pokemon was unable to be successfully stored in any of the boxes, this means the entire storage system is full. In this case, throw a NoStorageSpaceException. Recall that this exception does not take any parameters.

```
public T release(int box, int pos) throws OutOfBoundsException
```

First, check if box and pos are valid, meaning if they are in bounds. If either box or pos is not in bounds, throw an OutOfBoundsException and pass in the specified formatted String.

- The formatted String should be OUT_OF_BOUNDS_EXCEPTION, with the appropriate box and pos values filled in. Recall String.format() can help you with this.

Otherwise, box and pos are in bounds, so we want to get and return the Pokemon stored in Box at index box at index pos. Set the Pokemon at the appropriate position to be null to indicate that the position is now open, and return the Pokemon that was previously at the position.

Hint: Be sure to use any relevant methods you wrote earlier.

```
public void move(int boxFrom, int posFrom, int boxTo, int posTo)
    throws OutOfBoundsException
```

This method will swap the Pokemon at posFrom in boxFrom and posTo in boxTo. First, check if posFrom and boxFrom are valid. If either of them are invalid, throw an OutOfBoundsException and pass in a formatted

String.

- The formatted String should be `OUT_OF_BOUNDS_EXCEPTION`, with the appropriate box and pos values filled in. Recall `String.format()` can help you with this.

Then, check if `posTo` and `boxTo` are valid. If either of them are invalid, throw an `OutOfBoundsException` and pass in the specified formatted String.

- The formatted String should be `OUT_OF_BOUNDS_EXCEPTION`, with the appropriate box and pos values filled in. Recall `String.format()` can help you with this.

Otherwise, the parameters are valid, so swap the Pokemon at these locations.

```
public String getBox(int boxNumber) throws OutOfBoundsException
```

If `boxNumber` is not in bounds, throw an `OutOfBoundsException` and pass in a formatted String to the exception.

- The formatted String should be `OUT_OF_BOUNDS_EXCEPTION`, with the appropriate box value and with a pos value of 0.

Otherwise, return the String representation (`toString()`) of the `Box` at this `boxNumber`.

Simulator Class

Simulator.java

This class will simulate the catching and battling experience in the Pokemon universe using the classes you wrote above. In `Simulator.java`, we provided all the messages you need as String constants and provided a few other constants. To avoid misspellings and magic numbers, use the constants that we provide.

`Simulator` takes two required command-line arguments. The first argument selects your starter Pokemon. See `main()` below for more information on how to select the starter Pokemon. The second argument is the file name that contains a list of Pokemon that can be found in the wild.

After you complete the class and then compile and run `Simulator`, a battle with a rival Pokemon will take place right away. Then, the user will have the option to venture out into the wild to catch Pokemon or look in their Pokemon storage system.

There is a limited, pre-set group of Pokemon that can be found in the wild. This group is determined by an input file whose name is passed in as the second command-line argument. This input file is parsed in `parsePokemonFile()`. Once all the Pokemon in the input file have been encountered, there will be no more Pokemon in the wild, and an appropriate message will be displayed.

As stated before, the file is partially complete. Fill in the parts of the file described below and described in the TODO comments in the file to complete the class. Once you have completed the file, be sure to remove all TODO comments.

Methods

```
private static void handleBattle(Pokemon starter, Pokemon rival)
```

The most exciting aspect of the Pokemon world is battling. This method takes in two Pokemon as parameters and conducts a battle between `starter` and `rival`. The battle should be conducted in the following order:

- Print out the String constant `BATTLE_INTRO` using the name of the rival Pokemon followed by the name of the starter Pokemon. This is already done for you in the file.
- Get the damage from both Pokemon by calling `attack()`, and print out the String constant `BATTLE_MAIN` using the rival Pokemon's damage followed by the starter Pokemon's damage.
- Print out String constant `BATTLE_WIN` if Pokemon starter wins. Otherwise, print out String constant `BATTLE_LOSE`.
  - The Pokemon starter wins the battle if its damage is strictly larger than the damage of the rival Poekmon. This means that if the two Pokemon have the same damage, the starter Pokemon will lose the battle.

```
private static void handleWild(Pokemon wild)
```

Besides battling with another Pokemon trainer, Pokemon trainers spend time in the wild trying to catch new Pokemon. This method will handle wild Pokemon encounters.

Upon encountering a wild Pokemon, print out String constant `WILD_PROMPT` using the wild Pokemon's level and name. This is already done for you.

We have already written code for you that handles user input but have described it below to help you understand it. The user has two options, catch or run.

- If the user inputs `0`, the user wishes to catch the Pokemon. In this simulator, the user will successfully catch the Pokemon every time (a 100% catch rate!). Deposit the pokemon into the storage box, print out `CAUGHT_PROMPT` using the wild Pokemon's name, and terminate the function.
- If the user inputs `1`, the user wishes to run from the Pokemon. Print out `RUN_PROMPT` and terminate the function.
- For any other input, print out `UNRECOGNIZED_PROMPT` and reprompt the user for another input.

For this method, you simply need to add a catch statement. Catch the exception that might result from executing the code in the try statement. **Do NOT attempt to catch an exception of type `Exception`. You will lose points if you use `Exception`.** (Hint: Catch a type of exception that you wrote earlier.) Inside of the catch statement, print out the exception that was caught by using `System.out.println()`.

```
private static void handlePC()
```

You are responsible for filling in the TODOs in this method. We have described the overall behavior of this method below for you.

This method will handle the event where the user wants to interact with their PC, which is our Pokemon storage system. First, we print out `PC_PROMPT` to prompt the user for input. The user has three options, print out the contents of a box, move a Pokemon, or release a Pokemon.

For this section, you can assume that the user will always input an integer for their arguments.

- If the user inputs `0` as the first argument, the user wishes to print out the contents of the box. Parse the box number that they are passing in as an argument. Then, get the box that they're requesting for from the `storage` variable and print out its contents.
- If the user inputs `1` as the first argument, the user wishes to move a Pokemon. Parse the box and position arguments of the Pokemon they want to move and the box and position arguments of destination. Move the Pokemon accordingly.
- If the user inputs `2` as the first argument, the user wishes to release a Pokemon. Parse the box and position arguments of the Pokemon they want to release. Release the Pokemon accordingly.
- For any other input, print out `UNRECOGNIZED_PROMPT` and reprompt the user for another input.

```
private static List<Pokemon> parsePokemonFile(String filename)
```

You must write this entire method yourself. This method will read from a file, parse each line, and create a Pokemon object for each line. Here is an example of the file format:

```
016,Pidgey,10
025,Pikachu,10
035,Clefairy,15
```

The format of each line is the unique number for each Pokemon (dexNumber), Pokemon name, and level.

Initialize a new ArrayList and populate the ArrayList with new Pokemon objects using the information from each line of the file. Using the example above, you would create three Pokemon objects, add them to an ArrayList, and return that ArrayList.

Remember to catch any exception that may occur. Return null if an exception occurs. If a FileNotFoundException occurs, call on System.out.printf passing in a formatted string using the FILE_NOT_FOUND format string and the filename that was used.

```
public static void main(String[] args)
```

Check if the correct number of command-line arguments have been entered. As stated earlier, our program will take two command-line arguments.

Initialize the Pokemon storage system by setting `storage` to a new `PokemonStorageSystem` object.

Next, we will handle the first command-line argument by creating the appropriate starter Pokemon and rival Pokemon. Below is a table that displays what the starter and rival Pokemon will be based on the first argument. If the first command-line argument does not match any of the below choices, return to terminate the program.

choice	starter	rival
0	Charmander	Squirtle
1	Squirtle	Bulbasaur
2	Bulbasaur	Charmander

Recall that our constructors for our Pokemon can throw exceptions if they are given bad input (e.g. a level that is too high or a level that is too low). As a result, the code for creating the starter and rival Pokemon should be contained in a try statement. The catch statement should catch the types of exceptions that the constructors might throw. Inside of the catch statement, print the exception that was caught and return to terminate the program.

If the starter and rival Pokemon were created without any error, we set our party member to our stater Pokemon. This is already done for you.

Now we will conduct a battle between our starter and rival Pokemon. Call `handleBattle()` and pass in the appropriate arguments.

Next, we want to handle the second command-line argument by parsing the input file that contains the list of wild Pokemon. Set `allPokemon` to the return result of calling `parsePokemonFile()` with the file name, so that `allPokemon` is a list of all the wild Pokemon. We perform a null check on `allPokemon` afterwards to ensure that the parsing occurred correctly.

After parsing the input file, we will allow the user to decide what to do next. They can either go out into the wild or look in their PC.

- If the user inputs `OPTION_0`, they have indicated that they want to go into the wild. If there are not any more Pokemon in the wild that they have not encountered yet, we print out `EMPTY_WILD`. (Is there a local variable already declared for you that helps you keep track of the current wild Pokemon?) Otherwise, if there are still Pokemon in the wild to encounter, call handleWild() with the appropriate wild Pokemon. Be sure to update the local variable that keeps track of the current wild Pokemon!
- If the user inputs `OPTION_1`, they have indicated that they want to interact with their Pokemon storage system. Simply call `handlePC()` in this case.
- If the user inputs anything else, we print out `UNRECOGNIZED_PROMPT` as an error message.

We continue prompting the user for input until they signal EOF (end of file) by entering CTRL+D.

README

The following questions about provided files and general policies are graded for fair effort and completeness. Your file should be named `README.md`.

1. Instead of explicitly defining `Pokemon` as the type we're going to store in our `Position` class, we use type `T` instead. What are the benefits of doing this?
2. In `Box.java`'s `toString` method, we iterate through our List using a for each loop. Why is the type of the variable on the left hand side of the loop `Position<T>` rather than `Position`?

3. In your own words, explain the workflow of the `Simulator.java`.

Student Satisfaction Survey

Please fill out our student satisfaction survey. We are changing how we approach giving assignments and would like to hear about your experiences.

Style

We will grade your code style thoroughly. Namely, there are a few things you must have in each file / class / method (this includes the `README.md`):

1. File header
2. Class header
3. Method header(s)
4. Inline comments
5. Proper indentation
6. Descriptive variable names
7. No magic numbers
8. Reasonably short methods (if you have implemented each method according to specification in this write-up, you're fine). This is not enforced as strictly.
9. Lines shorter than 80 characters
10. Javadoc conventions (@param, @return tags, /** comments */, etc.)

A full style guide can be found here. If you need any clarifications, feel free to ask on Piazza.

Submission

Required Submission Files (13 files)

- `Box.java`
- `Bulbasaur.java`
- `Charmander.java`
- `MaxLevelException.java`
- `MinLevelException.java`
- `NoStorageSpaceException.java`
- `OutOfBoundsException.java`
- `Pokemon.java`
- `PokemonStorageSystem.java`
- `Position.java`
- `Simulator.java`
- `Squirtle.java`
- `README.md`

Start early and start often!