

Project 4: Space Colonies

Due Apr 11 by 11:59pm **Points** 100 **Available** Mar 22 at 7am - Apr 11 at 11:59pm 21 days

This assignment was locked Apr 11 at 11:59pm.

- It is an Honor Code violation to resubmit prior work from an earlier semester in this course.
- It is an Honor Code violation to submit work authored by someone else as your own.

You must include the following Virginia Tech Honor Code pledge at the very top of each Java file you write (separate from the class Javadoc comment). Include your name and PID as shown, to serve as your signature:

```
// Virginia Tech Honor Code Pledge:
```

```
//
```

```
// As a Hokie, I will conduct myself with honor and integrity at all times.
```

```
// I will not lie, cheat, or steal, nor will I accept the actions of those who do.
```

```
// -- Your name (pid)
```

Project 4 Introduction

SpaceColo2018S 0324



For this project, your job is to write a program which assists the process of placing applicants in homes on colonies in outer space.

Imagine a scenario: The year is 2055. You work in the newly opened Galactic Emigration Office for planet Earth. Three new planets are ready for human occupation, and the time has come to make assignments. You are given a list of applicants to live on these new colonies, and information on each planet. Your job is to write a program that can go through this list of applicants in order, and either place each person in their new home or reject them, according to certain criteria.

As a part of this project, our goal is to simulate and visualize how these decisions are made for each unique applicant. A number of scenarios arise. Your job is to think through these scenarios before you begin the implementation. This project aims to help you understand and develop systems where data flows between a number of classes and different classes communicate and collaborate with each other. Below is the description of the basic scenario, try to get the big picture first and then delve into the details of each part.

First, each person's application contains the following data: The applicant's name, their scores in three skills (MEDICINE, AGRICULTURE, and TECHNOLOGY), and an optional field to request which colony they want to live on. In addition, each planet has a name, representations of the current population, a maximum capacity, and a minimum requirement in each of the three skills.

If the person has a valid colony request (the person wants one of the three planets given), they **MUST** either be placed in **THAT** colony or rejected altogether. If there is room on the requested planet, the program will need to check the person's scores against the planet's minimum requirements, then make the decision to allow the user to accept or reject their application. If the person has no colony request, the user should be given the option to accept placement on the colony with the most available space whose minimum requirements are suitable.


To track this movement of people, we are going to use several data structures that you've learned over the course of this semester, as well as several other topics such as interfaces, encapsulation, Window, and File I/O.

For this last guided project, we want you to try to figure it out yourself as much as possible. We will start by giving you the broadest overview of the project, and slowly fill in the gaps. It is strongly recommended that you always read through all the instructions, before you start coding.

As you write each class, write its Junit tests. It is recommended that you use Test Driven Development, which requires you to code the tests before you write the class. Fully test each class before you move on to the next one, (i.e., practice piecemeal development). If you don't test from the ground up, it will be difficult to tell where a bug is coming from when combining many classes and functions.

WebCat will not grade for test coverage of ProjectRunner, SpaceWindow and ColonyReader. It is very important that these classes still work but points will not automatically be deducted for test coverage.

While grading, GTAs will focus on running the code and majority of your points will be graded on the GUI (Graphical User Interface)

Input will come from two files, a file with a list of three planets and a file with a list of applicants. Both txt files contain values separated by commas. Download sample files here: [SpaceColoniesInputFilesCleanAndErrors.zip](#) 

The format of the applicant input file is formatted accordingly:

<Applicant Name>, <Agriculture Score>, <Medical Score>, <Technical Score>, <Planet Preference>

And will look something like the following:

Marva Buel, 1, 2, 1, Planet1

Dian Cinnamon, 3, 1, 5, Nowhere

Scottie Gebhardt, 1, 2, 2,

Louella Voorhies, 3, 3, 1, Planet1

Mandy Vaugc 4, 1, Planet3

Their names are always followed by their skill values in alphabetical order (agriculture, medicine, then technology).

For the planet preference field, it will either contain the name of a planet in the program, some other string, or be empty (no space after comma).

The format of the applicant input file is formatted accordingly:

<Planet Name>, <Agriculture Minimum>, <Medical Minimum>, <Technical Minimum>, <Capacity>

And will look something like the following:

Hokies, 5, 3, 5, 10

Wahoos, 2, 5, 5, 8

YellowJackets 3, 3, 5, 9

The planet names are always followed by their minimum skill values in alphabetical order (agriculture, medicine, then technology), then the planet capacity.

The planets should be numbered 1, 2, 3 in the order in which they occur in the input file. To correspond with our test code, store the planets in sequence in planets[1], planets[2] and planets[3].

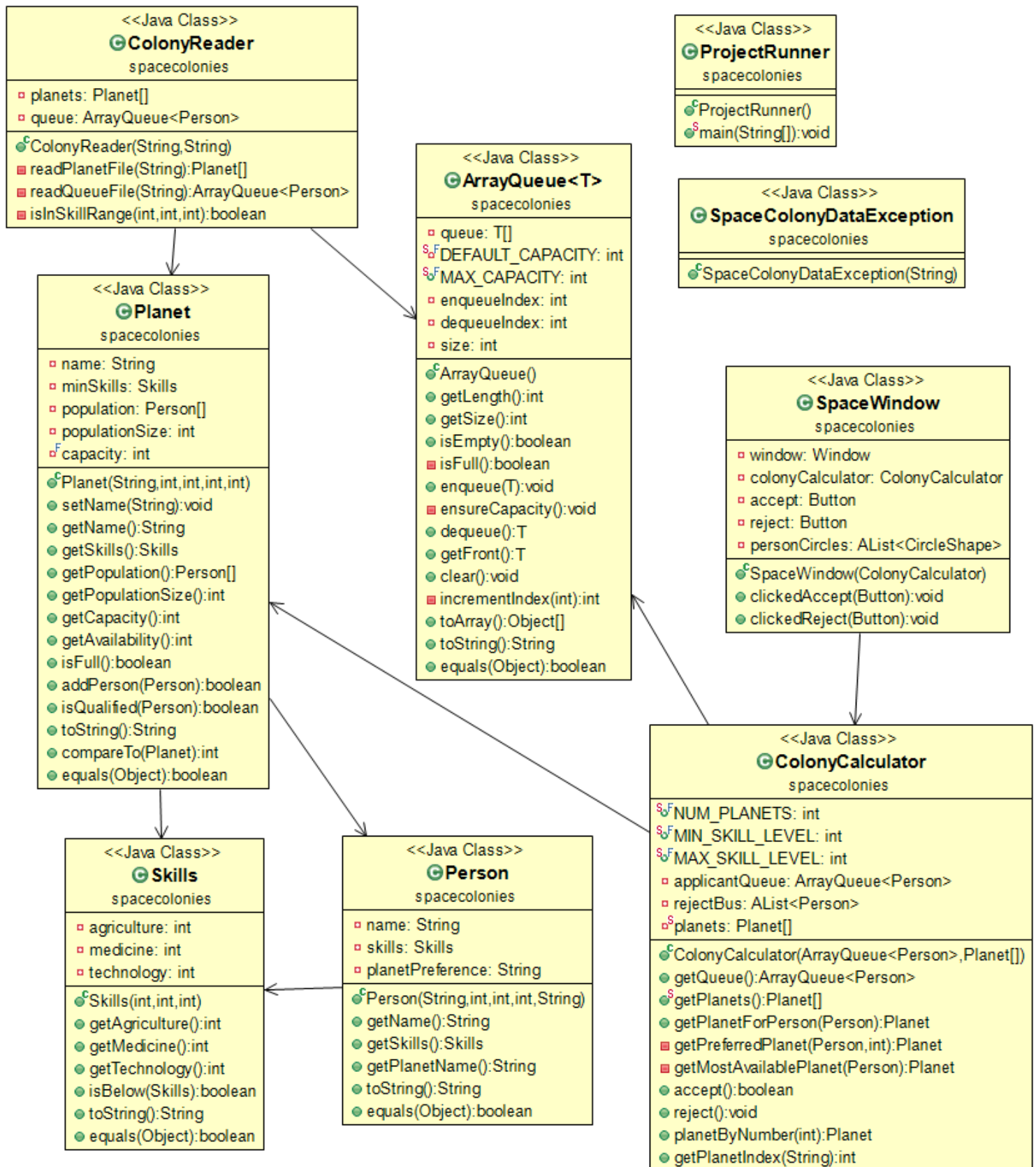
You can use <http://listofrandomnames.com/> to build your own input.

If `NumberFormatException`s are thrown during input, it is a likely indication that input file format is not the same as what the code is expecting.



Project 4 Class Overview

UML



Overview of Classes

Project name: SpaceColonies

package name: spacecolonies

An overview of the classes is given in this section, step-by-step implementation suggestions for each class is provided after the General Guidelines below. The diagram does not represent when the return type is generic.

Person: These objects contain a string, for a person's name, a skills object, and a String representation of their planet preference.

Skills: These objects contain three ints for their skills(on a scale of 1 to 5).

Planet: These objects contain a string, for the planet's name, three ints for their minimum skill requirements (on a scale of 1 to 5), an array of Person objects for current planet population, an int for storing the current population size, and a final int for the maximum allowed capacity of the planet.

ArrayList: Import this list class from CarranoDataStructuresLib. You do *not* implement it.

ArrayQueue: This data structure implements QueueInterface with a circular array implementation. It provides default queue behavior, such as enqueue, dequeue, getFront, and isEmpty.

ColonyCalculator: This object handles all the major calculations and decision-making for the program. It is in charge of handling accept and reject instructions and checking that all requirements for a person are met before they are added to a planet. It works together with SpaceWindow.

SpaceWindow: This object is the front end. Here we build our window, its buttons, and render the Planets and the queue of applicants on the window in a meaningful way.

There is also an UI Example Guide at the end of the page that can help you to build your SpaceWindow.

SpaceColonyDataException: This will be thrown if data is incorrect in the input files.

ColonyReader: The ColonyReader parses the input data from two text files. It generates the planets and queue of applicants based on one file of comma separated values about applicants and the other about each planet. Then it gives SpaceWindow this queue in order to tie everything together.

ProjectRunner: The ProjectRunner class begins the program by creating a QueueReader and telling it which file to look at.

General Recommendations

Order of Implementation

Be mindful of the order in which you implement the classes. Write and run tests for each class as you go. Some classes depend on each other, so be sure one is correct before implementing the next. For example, ColonyCalculator depends on both Person and Planet. Once you have your JUnit tests working, implement parsing an input file and test more data.

Remember, WebCat will not grade for test coverage of ProjectRunner, SpaceWindow and ColonyReader. It is very important that these classes still work but points will not automatically be deducted for test coverage.

Fields

- Should typically be private. If they are final static literals, which could not be changed by giving public access, make them public.
- Make them final if you expect them to be set once and never again.

- Make them static if you are pre-declaring a literal value, (constant), for use in a class, such as information which will always be the same for all objects of a class.

Getters and Setters

- We expect your getters to be declared as “get”+<fieldname>().
- Only provide setters if you expect those fields will need to be changed, aka if they correspond to fields which are not final and are mutable.

Methods

- Declare methods public when a user of your class will need access to it.
- Declare helper methods private as much as possible. If it's only used inside of the class, make it private.
- If you're copying and pasting code, make a method.
- If you're writing complicated code, which could be error prone, make it a method and test it separately. Examples include large logical statements, or edge cases. Note that you can have multiple test methods for one method.

equals()

For this project, different instances of a class are equal if their fields are logically the same. For example, if you instantiate two different Persons, but give them the same name, skills, and planet preference, they are equal.

You are expected to write an equals() method for most classes. This is a good programming habit. It is common that you will want to use equals() to debug, so you should preemptively have it in your code. Not having equals() implemented can easily cause hard to detect silent bugs. It is easy to write code that depends on equals without realizing it's not implemented and is actually testing for identity which seems like it works but is logically incorrect. Be sure to correctly code your equals() methods.

toString()

If you're building a complex string which is dependent on multiple facets of an object instance, use a **[StringBuilder](http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html)** (**[Links to an external site.](http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html)**)

(**<http://docs.oracle.com/javase/7/docs/api/java/lang/StringBuilder.html>**). StringBuilder is much lighter on memory and will make your code faster. When you operate only on String objects, since they are immutable, every time you change them a new string is instantiated in memory. That can get to be expensive in terms of memory usage!

You are expected to write a toString() method for most classes. This is also a good programming habit. It is common that you will want to use toString() to debug, so you should preemptively have it in your code.

The summary of the toString() output for each class is as follows:

Skills: A:2 M:5 T:4

Person (with valid planet preference): Jane Doe A:3 M:2 T:1 Wants: Nars

Person (without valid planet preference): No-Planet Jane Doe A:2 M:5 T:4

Planet: Nars, population 5 (cap: 10), Requires: A >= 3, M >= 2, T >= 1

ArrayQueue: [Jane Doe A:3 M:2 T:1 Wants: Planet Number 1, No-Planet Jane Doe A:2 M:5 T:4]

Guard against NullPointerException

If you ever call a method on an instance of an object, be aware that you may need to check whether the object is null and handle it as a special case. A special case where a method returns null can also provide useful information about the state of the object, such as a queue returning null when empty.

Include a statement similar to the following before you use parameters if you would like to avoid NullPointerExceptions.

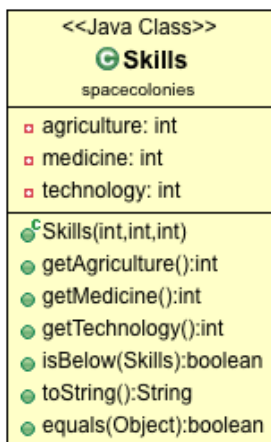
```
if(parameter == null) { ///handle special case }
```



Project 4 Implementation Breakdown

Implementation Guidelines Class-by-Class

Skills



public Skills(int ag, int med, int tech)

The Skills constructor should take 3 parameters, one for each skill category, in alphabetical order. The values can be from 1 to 5.

public boolean isBelow(Skills other)

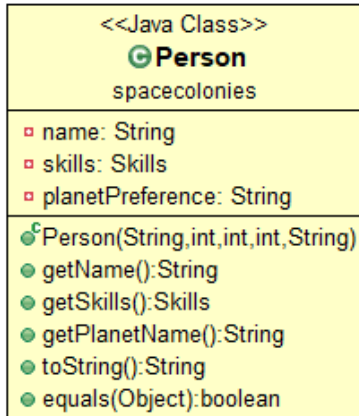
Compare a given Skill "other" to "this" Skill. Returns True only if "this.agriculture" is less or equal to "other.agriculture" AND "this.medicine" is less or equal to "other.medicine" AND "this.technology" is less or equal to "other.technology".

This method can be used for checking if a person's skill set is accepted based on a planet's minimum required skillset.

public boolean equals(Object obj)

Two Skills objects are equal if all three fields, agriculture, medicine, and technology, are equal.

Person



public Person(String name, int agri, int medi, int tech, String planet)

The Person constructor should take 5 parameters, the person's name, their skills levels in all these areas and their planet preference. This data should be listed on each line of the applicant input file.

public String getPlanetName()

This method should return the name of the planet preferred by the Person.

public String toString()

Using a StringBuilder, concatenate the name, the first letter for each skill (in alphabetical order) and a colon, then the skill value. Then, if the person has a planetPreference String that has a length greater than 0, add "Wants: " and the planet name. Note the space after the : in "Wants: ".

Sample output: "Jane Doe A:3 M:2 T:1 Wants: Nars" or "No-Planet Jane Doe A:2 M:5 T:4"

public boolean equals(Object obj)

Two Person objects are considered equal when their name, skills, and planet preference value is the same. You will need to write your own equals method.

ArrayQueue<T> implements QueueInterface<T>

<<Java Class>>	
 ArrayQueue<T>	
spacecolonies	
<ul style="list-style-type: none"> queue: T[] DEFAULT_CAPACITY: int = 10 MAX_CAPACITY: int = 100 enqueueIndex: int dequeueIndex: int size: int 	
<ul style="list-style-type: none"> ArrayQueue() getLength():int getSize():int isEmpty():boolean isFull():boolean enqueue(T):void ensureCapacity():void dequeue():T getFront():T clear():void incrementIndex(int):int toArray():Object[] toString():String equals(Object):boolean 	

The queue can initially hold DEFAULT_CAPACITY objects. If more objects need to be added, then the size can be expanded until it reaches MAX_CAPACITY objects. If MAX_CAPACITY is exceeded, then throw an `IllegalStateException`.

The parties waiting to ride need to be tracked in order, since they are given to us in the order which they have arrived in line,(i.e., FIFO order). As you've recently discussed in class, the best way to track people in line is to use a queue.

See your textbook for information on how to implement Circular Array ArrayQueues! Notice this implementation keeps track of the enqueueIndex (AKA backIndex) and dequeueIndex (AKA frontIndex) and uses those to determine the number of items in the queue. A field for the size can be maintained to avoid the calculation but is unnecessary and with each change to the queue must be updated which increases the risk of error. This project requires the implementation of the methods `size()`, `equals()`, `toArray()` and `toString()` in addition to those described in the text.

A full ArrayQueue will have one empty slot. This null is to keep your dequeue and enqueue indices from overlapping. They only overlap when the queue is empty. When you create your array, build it using `DEFAULT_CAPACITY + 1`. This way, the back end array will have an actual size of 11, but will be only able to hold 10 elements.

An ArrayQueue implementation needs to correctly handle the cases where the indices wrap around. Be sure to write test cases for such scenarios, and your textbook has useful examples.

private void ensureCapacity()

This optional helper method can be used to upgrade the length of the queue when the queue is full. The new length is twice as large as the old size. Note that size is different from length. So a queue of length 11 will be full when the size is 10, and after upgrading the length in `ensureCapacity()`, the new length will be 21 with the capacity to hold 20 elements.

private int incrementIndex(int index)

This is an optional helper method, but will help you with the circular queue wrapping logic. Anywhere you increment an index in the array, you should use this method or similar modular arithmetic.

```
return ((index + 1) % queue.length);
```

public void clear()

Think of clear as hitting the reset button on your object. You can also think of the clear method as a way to re-call the constructor. When you implement clear, try to match your constructor, (i.e., clear should result in the object being in the same state as a newly constructed object).

Example: Say you were clearing a LinkedStack like in project 3. Instead of calling remove over and over until it is empty, simply set your fields to their default values. This executes faster, and has the same logical result.

public Object[] toArray()

Queues generally do not have a method that lets us cycle through the elements like we can in a list. Implementing a toArray() method gives client code the option of accessing the data in the queue without interfering with the integrity of the queue.

Your toArray() cannot just return the underlying array from this. The returned array needs to have its entries start at index 0, which might not be true of the circular array. The returned array should only be as large as this.size, meaning it has no extra empty slots.

Throw an EmptyQueueException() when the queue is empty.

public String toString()

The toString() method will also need to iterate through the contents of the queue. Concatenate the toString() of each Object in the ArrayQueue separated by a comma and space, except for the last object in the queue. StringBuilder is especially efficient in these situations where we are building a string in a loop. Surround the whole thing with brackets, []. If the queue is empty, return "".

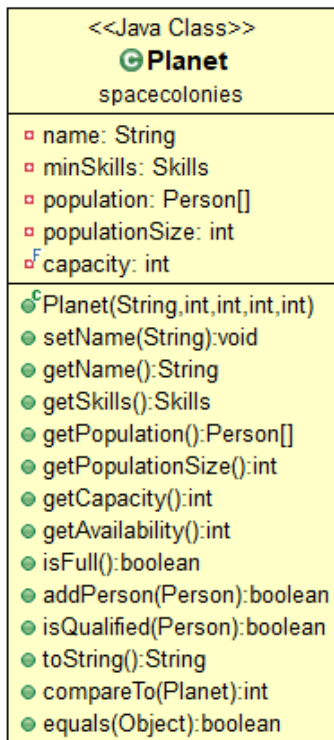
For example: [Jane Doe A:3 M:2 T:1 Wants:Nars, No-Planet Jane Doe A:2 M:5 T:4]

public boolean equals(Object obj)

For two ArrayQueues to be equal, they have to contain the same elements in the same order. With a circular queue implementation, it is possible the elements are not stored in the same indices in each underlying array. So, compare the sizes of the two ArrayQueues and then iterate over the contents. For example:

```
for (int i = 0; i < size(); i++) {  
    T myElement = queue[(frontIndex + i) % queue.length];  
    T otherElement = other.queue[(other.frontIndex + i) % other.queue.length];  
    if (! myElement.equals(otherElement)) {  
        return false;  
    }  
}
```

Planet



public Planet(String planetName, int planetAgri, int planetMedi, int planetTech, int planetCap)

The Planet constructor should take 5 parameters, the planet's name, the minimum skill level required in all three areas and the maximum capacity for that planet. This data should be listed on each line of the planet input file.

public int getAvailability()

This method should return the number of available places left in the planet, by using the planet's capacity and current population size.

public boolean isFull()

This method should return true if the planet's population has reached max capacity.

public boolean addPerson(Person newbie)

This method will attempt to add a Person to the Planet. It will need to check two things. First, that the colony has available space for this applicant. Secondly, it will need to determine whether the applicant is qualified to live on the colony.

HINT: Consider using a helper function (see isQualified(Person applicant) in the given UML).

public String toString()

Using a StringBuilder, concatenate the name, "population" followed by the current population size. Next concatenate the capacity of the planet, and then the required skill values in the following format:

"Caturm, population 5 (cap: 10), Requires: A >= 3, M >= 2, T >= 1"

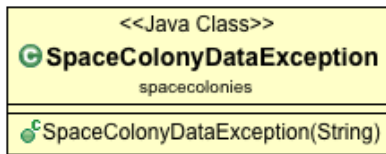
public boolean equals(Object obj)

Two planets are equal if all 5 their input fields are equal and populationSize is equal.

public int compareTo(Planet other)

Planet should implement the Comparable Interface and the compareTo method should order the planets based on availability. A planet with more available slots will be greater than one with less. Note that this is slightly different than how equals works. compareTo will return 0 if availability is equal regardless of whether the rest of the planet attributes are the same.

SpaceColonyDataException

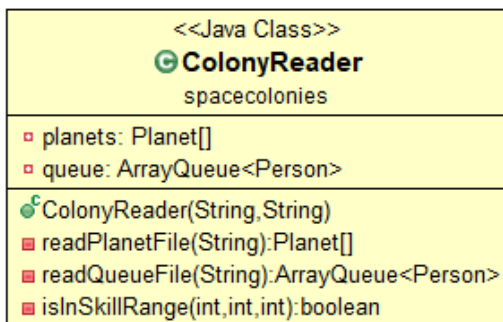


This should extend the standard Java Exception

public SpaceColonyDataException(String string)

The constructor should call super and pass it the string message.

ColonyReader



Place the input files in your root directory. When you look at your project folder's structure, the input file should be on the same level as the src and bin folders. After adding it, you may need to refresh your project file structure by hitting F5 or right-clicking the project and choosing Refresh.

The ColonyReader class will read the input and begin the SpaceWindow. In the constructor, you will call private methods to complete parsing the parsing and then create a ColonyCalculator for processing. Pass your ColonyCalculator into another class, SpaceWindow, to run your program.

You can download some small sample input files from above, and place them in your project folder. You need to parse the text files and extract data from them. The structure of the files is described in the project introduction. The ColonyReader Class will read from the input file It is responsible for parsing the input about the planets and the people applying to live on them.

public ColonyReader(String applicantFileName, String planetFileName)

Send the applicant input file to readQueueFile to populate an ArrayQueue<Person> for the colony and send the planet input file to readPlanetFile to populate an array of planets for the colony.

Instantiate a new SpaceWindow with the recently filled colony as its parameter.

private Planet[] readPlanetFile(String fileName)

Declare and instantiate a local array of planets that will store the planets in sequential order in slots 1, 2 and 3.

Declare a new Scanner named file and instantiate it to be a new Scanner, with a new File as its parameter. Give your new File the String fileName parameter. You do not have to use a try catch statement to catch a FileNotFoundException, you can just let it be thrown.

If the filename was correct, you can begin parsing the file. Loop through it using hasNextLine(). Only read in the first 3 Planets.

You can use String's split() function to parse the lines. The delimiter is "," but if you want to also account for spaces you can use a regular expression ", *" (note the space). This will ensure your numerical skill values can easily be converted with Integer.valueOf(String).

If there are not 5 comma separated values on the line, then throw a ParseException.

If the skills are not between 1 and 5, then throw a SpaceColonyDataException.

If there are less than 3 planets, then throw a SpaceColonyDataException.

private ArrayQueue<Person> readQueueFile(String fileName)

Declare and instantiate a local ArrayQueue<Person>, this will store the data in the input file.

Follow the same strategy as above in readPlanetFile for using a Scanner object and parsing the lines.

If the skills are not between 1 and 5, then throw a SpaceColonyDataException.

Hint: You may get an empty string if you use your scanner's nextLine() method to parse the people's names. That's because after parsing the previous integer, the scanner is still "looking" at the end of the previous line. Call nextLine() a second time to get their full names.

Hint: You can handle the "no preference" case as you choose, one suggestion is to use an empty String.

Reference your previous labs and textbook for File I/O guidance as needed.

isInSkillRange(int num1, int num2, int num3)

This function should return whether or not all of the integers it is passed (num1, num2, and num3) are between the minimum and maximum possible values for a skill.

ColonyCalculator

<<Java Class>>	
ColonyCalculator	
spacecolonies	
S ^F	NUM PLANETS: int = 3
S ^F	MIN SKILL LEVEL: int = 1
S ^F	MAX SKILL LEVEL: int = 5
□	applicantQueue: ArrayQueue<Person>
□	rejectBus: AList<Person>
S ^S	planets: Planet[] = new Planet[NUM PLANETS + 1]
⬢	ColonyCalculator(ArrayQueue<Person>, Planet[])
⬢	getQueue(): ArrayQueue<Person>
S ^S	getPlanets(): Planet[]
⬢	getPlanetForPerson(Person): Planet
■	getPreferredPlanet(Person, int): Planet
■	getMostAvailablePlanet(Person): Planet
⬢	accept(): boolean
⬢	reject(): void
⬢	planetByNumber(int): Planet
⬢	getPlanetIndex(String): int

Overview

This class does the major calculations for the program. It handles the accept/reject logic and contains the queue of applicants and the Planet objects.

You should use "NUM_PLANETS", "MIN_SKILL_LEVEL", and "MAX_SKILL_LEVEL" in your ColonyReader or other classes as need to avoid hard coding.

public ColonyCalculator(ArrayQueue<Person>, Planet[])

The constructor. An "IllegalArgumentException" will be thrown if the input ArrayQueue is null.

public Planet getPlanetForPerson(Person nextPerson)

This method will determine if the next applicant can be accepted to a planet, you may (you should) want to write additional helper methods for this functionality (such as getPreferredPlanet and getMostAvailablePlanet). First, ensure that the applicant line is not empty. Then you will need to determine the person's planet preference. If they have a preference, then check that the planet is not full and the applicant matches the eligibility requirements. If they would be accepted to that planet, then return that planet, otherwise return null.

If the Person passed as a parameter to this method is null such as the method being called getPlanetForPerson(null) it should return null.

If they do not have a valid planet preference, you should try to place them to the planets in the order of availability. A suggested approach for this to make a deep copy of the array of planets and then use Arrays.sort() to order the copied array by availability. If planets have the same availability, the person will be assigned to the planet with a higher number, so if Planet1 and Planet3 both have the highest number of slots available, the applicant without a preference will first be applying to Planet3. Next check that the planet is not full and the applicant matches the eligibility requirements. If they would be accepted to that planet, then return that planet, otherwise repeat the process for the planet with the next most number of available slots. If they would not be accepted to any of the planets, return null to indicate that they cannot be accepted anywhere.

public boolean accept()

This method will attempt to accept the next applicant. First, ensure that the applicant line is not empty. Then you will need to determine the person's planet preference with `getPlanetForPerson` in order to send them to the correct planet. Add the person to their preferred planet and then deque them from the applicant queue.

public void reject()

When `reject` is called, the next applicant in line should be put on the bus back to skill training school. Remove the `Person` from the line and add them to an `AList<Person>` that will represent the bus.

public Planet planetByNumber(int planet)

Return the `Planet` object for the given number (1, 2, or 3). For any other number, return null.

public int getPlanetIndex(String planet)

Return the `int` representation for the given `String` (planet name). If the given `String` is not a name for any of the three planets, return 0.

public ArrayQueue<Person> getQueue()

This method should return the `ArrayQueue` of applicants.

public Planet[] getPlanets()

This method should return the array of planets.

ProjectRunner

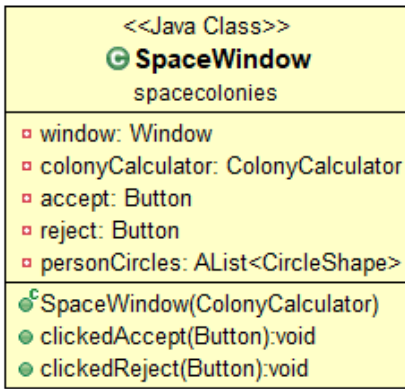


Once again, you will use a `ProjectRunner` class to begin your program. Typically, your main method will be very small. Instead of parsing input in the main method, it is better practice to place it in a `Reader` class, in this case `ColonyReader`.

public static void main(String[] args)

If two arguments have been provided, we will use those as our input filenames. Otherwise, instantiate a `ColonyReader` class and provide the default "input.txt" and "planets.txt" filenames. The main method should not catch the exceptions, just let them be thrown.

SpaceWindow



The private methods and some of the private variables are hidden on the UML, add your own private methods and private variables as needed.

//This class diagram leaves space for where you will write and name your own private variables and methods.

SpaceWindow is responsible for the visualization of the applicants moving through the line, which you can see in the Youtube video at the top of this page. A circle is used to represent every Person in the queue. The color of the circle represents the applicant's planet preference and will **match the corresponding Planet's representing Shape**.

The planets will be represented with squares, that are filled to show how much remaining space is left in the planet.

You will be in charge of determining how to best decompose this problem into public and private methods, field and local variables, etc. The public methods and private field variables shown in the UML are the only methods and variables you are required to have. This section will provide some hints for you about how to split up the work.

Overview

In this class, we build a Window and title it 'Space Colony Placement'. On the south side, we will have a Button named accept, and a Button named reject. The reject button is always enabled. If the person at the front of the line would be admitted to a planet then the accept button is enabled. When the accept button is clicked, then the person is moved out of the queue and onto a planet. Every time reject is clicked, the Person at the front of the line will be removed from the line and put on the bus back to skill training school. Most of these calculations will be handled by ColonyCalculator, so it's SpaceWindow's job to update the visuals accordingly.

You won't be able to run this class and check your code until you complete the QueueReader and ColonyCalculator classes.

There is also an UI Example Guide at the end of the page that can help you to build your SpaceWindow.

Field Variables

At a minimum, the class will need to keep track of these objects:

- A Window which displays the visuals.
- A ColonyCalculator, which handles the accept/reject logic and contains the queue of applicants and the Planet objects.
- Two buttons, one which says "ACCEPT" and one which says "REJECT".

You will most likely need more field variables than these. If you find yourself creating many objects of the same kind, such as CircleShapes to display the applicants, then having an AList or an array of that object may be helpful, such as an AList called personCircles which contains a CircleShape for each Person displayed in the queue.

As you code, consider the pros and cons of having field variables accessible to the whole class versus local variables. If you need to access a variable from multiple methods, such as your queue, then having a class variable is a good idea. If you only need to interact with the variable temporarily or a single time, such as the vertical separator Shape, then a local variable might be fine.

Constants

It might help your readability if you keep some constants to determine placement and size of Shapes on the screen. For example, instead of setting a Shape to have a height of `10*planet.getPopulationSize()`, you can have `PLANET_HEIGHT*planet.getPopulationSize()` so that any time you need to change the factor of your radius, you only have to change where you set your constant instead of changing every single time you typed "10".

```
public static final int QUEUE_STARTX = 100; //The horizontal position which starts your queue
```

```
public static final int QUEUE_STARTY = 150; //The vertical position which starts your queue
```

```
public static final int CIRCLE_SIZE = 10; //To use to size each Person circle.
```

These are only examples and you should make constants as you see fit. Even if you don't need a constant for a value, it's best to have a variable of some sort rather than hardcoding the number.

public void clickedAccept(Button button)

When accept is clicked, first check to ensure that there is a valid, nonempty queue of riders. Then you can attempt to use ColonyCalculator's `accept()` method to accept the next Person in line. If the applicant is successfully accepted, the window should update to represent this. The accept button should be disabled if the application can not be accepted, in order to force the user to click reject.

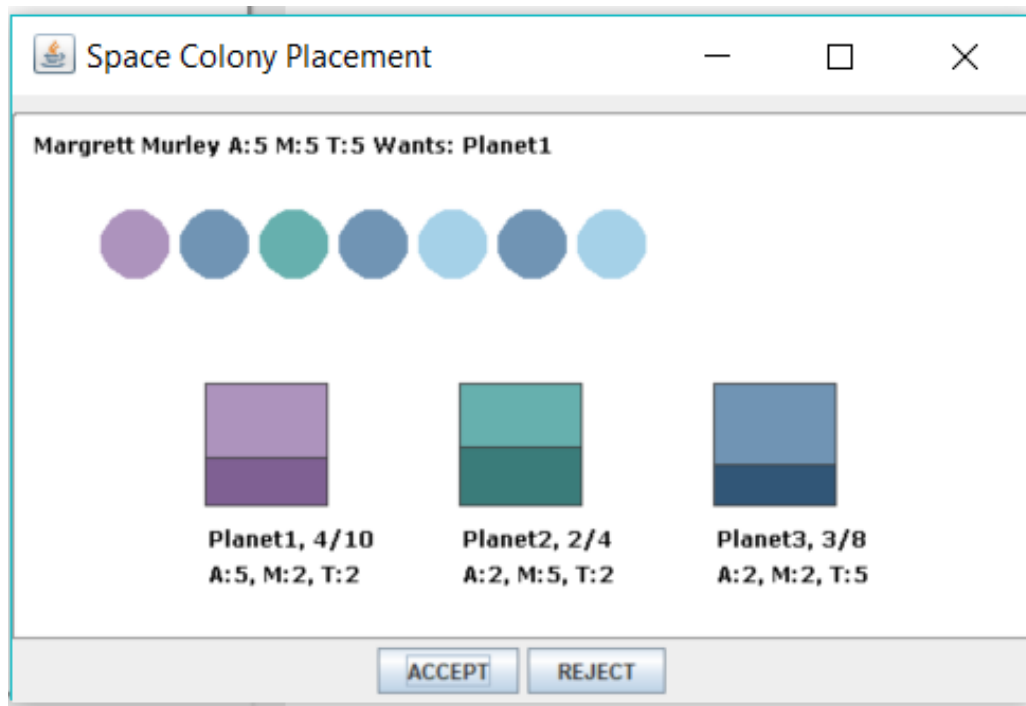
public void clickedReject(Button button)

When the user clicks reject, first you should use the corresponding method in ColonyCalculator to reject the applicant in the back-end. Then, the accept button should be enabled (in case it was disabled by `clickedAccept()` previously), and the visuals updated to represent the Person's absence

Note that there is no visualization of the rejected applicants in this project. They just disappear from the screen, and the queue visualization updates.

Also note that the user can choose to reject any applicant, regardless of eligibility.

Example visualization



Drawing Shapes in the window

As the program progresses, it will need to render and update Shapes in the window to represent the underlying data. There are two common approaches to changing how a window updates: keep track of the Shapes in the window and change their position, color, size, and remove them as needed, or erase all objects from your screen and put everything back on in its correct position. Be sure to read the [Class API on GraphWindow](http://courses.cs.vt.edu/~cs2114/Spring2016/GraphWindow/JavaDocs/) (<http://courses.cs.vt.edu/~cs2114/Spring2016/GraphWindow/JavaDocs/>) to be aware of the methods at your disposal.

In either case, it might be helpful to have helper methods manage some of the updating, especially updating that needs to be done more than once or in more than one case.

The Shapes you will need to have on your window can be seen in the Youtube video and in the screenshot above. Note that most shapes will need to be updated often.

The CircleShapes representing the applicants should be equal in size, and should be colored based on their planet preference. This color needs to match the corresponding Planet shape. They should be centered vertically when compared to each other (in a horizontal line).

The colored Shape representing how full the Planet is should be proportional to the number of spaces open and should fill from the bottom. You might find it useful to use two shapes for this bar and either overlay one on the other or update them both to meet at the right spot. (If the wrong shape is in front, Window provides `moveToFront(Shape shape)` and `moveToBack(Shape shape)` methods.).

Color Note

You may use any colors for this GUI, provided they are clearly distinguishable. `java.awt.Color` provides several static colors, which you can use for this project. For more complex colors, use the constructor `Color(int r, int g, int b)`. To get the RGB values for a color, you can use a tool such as this: http://html-color-codes.info/#HTML_Color_Picker (http://html-color-codes.info/#HTML_Color_Picker)

The colors used in the example implementation are:

No preference: 165, 209, 232

Planet 1: 173, 147, 189

Planet 1 filled: 127, 96, 147

Planet 2: 102, 176, 174

Planet 2 filled: 58, 124, 122

Planet 3: 112, 148, 180

Planet 3 filled: 49, 86, 119

End of the simulation

Once the queue is empty, and all applicants have been accepted or rejected, you've completed your program! Erase everything from the screen, display "All Applicants Processed - Good Work!" where your Queue's CircleShapes were, and disable both your accept and reject buttons.

Optional: Additional Implementations

You are not expected to implement the following ideas for this project, but you should be starting to think more and more about the design issues of your projects. Here are some concepts to consider:

When the simulation finishes, it would be useful if we could store and display the results in some way. What would it require to display statistics in ShapeWindow, such as the final planet rosters and percentage of applicants accepted? Or what about saving the same information to a file?

UI Example

Check out the UI examples below to help you build your SpaceWindow:

[Project4 UI Examples](#)



Submission

Finally, Submit to WebCat! Visit office hours if you're still having trouble. Focus on getting these in order...

- Test Coverage (make your tests pass)
- Problem Coverage (Address hints given)
- Code Coverage (Test more code to get more hints)

Submit your work to Web-CAT by the due date and time. You can use the Submit Assignment feature in Eclipse to submit your project directly without leaving the Integrated Design Environment (IDE). You may submit as many times as you'd like before the project is due.

✓ Grading Rubric

GUI

Web-CAT

Program Grading Rubric

- 50 pts according to [Project Grading Rubric.](#)