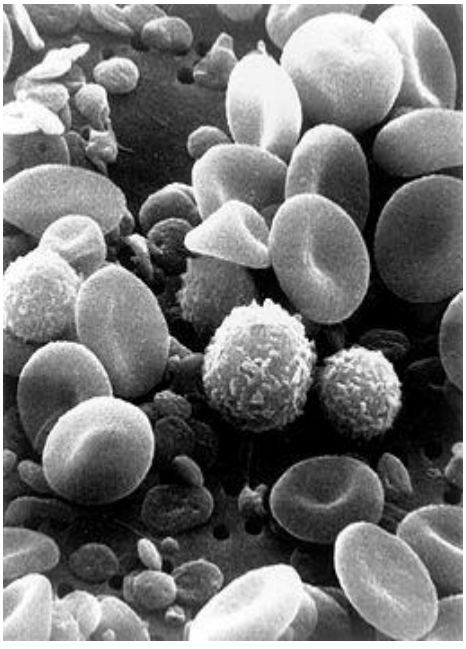
Programming Project 1

In this project, you'll be writing some code to create various classes that model interactions between various entities in human blood, such as white blood cells, viruses, and bacteria. You will use object-oriented programming to explore message passing between objects of various types. In particular, after successfully completing this project, you'll have met the following goals:

- An understanding of inheritance of private, protected, and public attributes and methods.
- An understanding of abstract classes.
- An understanding of enumerations (which we will learn about here for the first time).
- An understanding of static elements of classes.
- An understanding of method overriding.
- An understanding of the `super` concept.
- An understanding of primitive arrays in Java.

Once you have completed this project, you will be ready to take **Assessment 4**. You will be uploading and using your own code for Assessment 4, so make sure to spend time commenting your code and making it clean and easy for you to work with.

Introduction



On average, adult humans have about five liters of blood, which is composed of mostly *plasma*, as well as several types of red and white blood cells, among other things. In addition to serving the body's oxygen and glycogen needs, our blood is also responsible for localizing and eliminating various kinds of pathogens, though our *immune system*. How well our immune system functions against particular pathogens depends on many factors, including the number and ability of our white blood cells to neutralize various intruders. For people with compromised immune systems, it is important to be able to understand the balance between the immune system and these pathogens in order to more effectively target treatments.

In this project, we will attempt to model a small part of the immune system activity by running simulations on various blood samples. We'll be able to model and measure what happens when different ratios of different types of white blood cells and pathogens exist in the blood at the same time: can the immune system defeat all the intruders, or is the patient at risk of losing to the disease? Although our models are going to be rudimentary and limited in

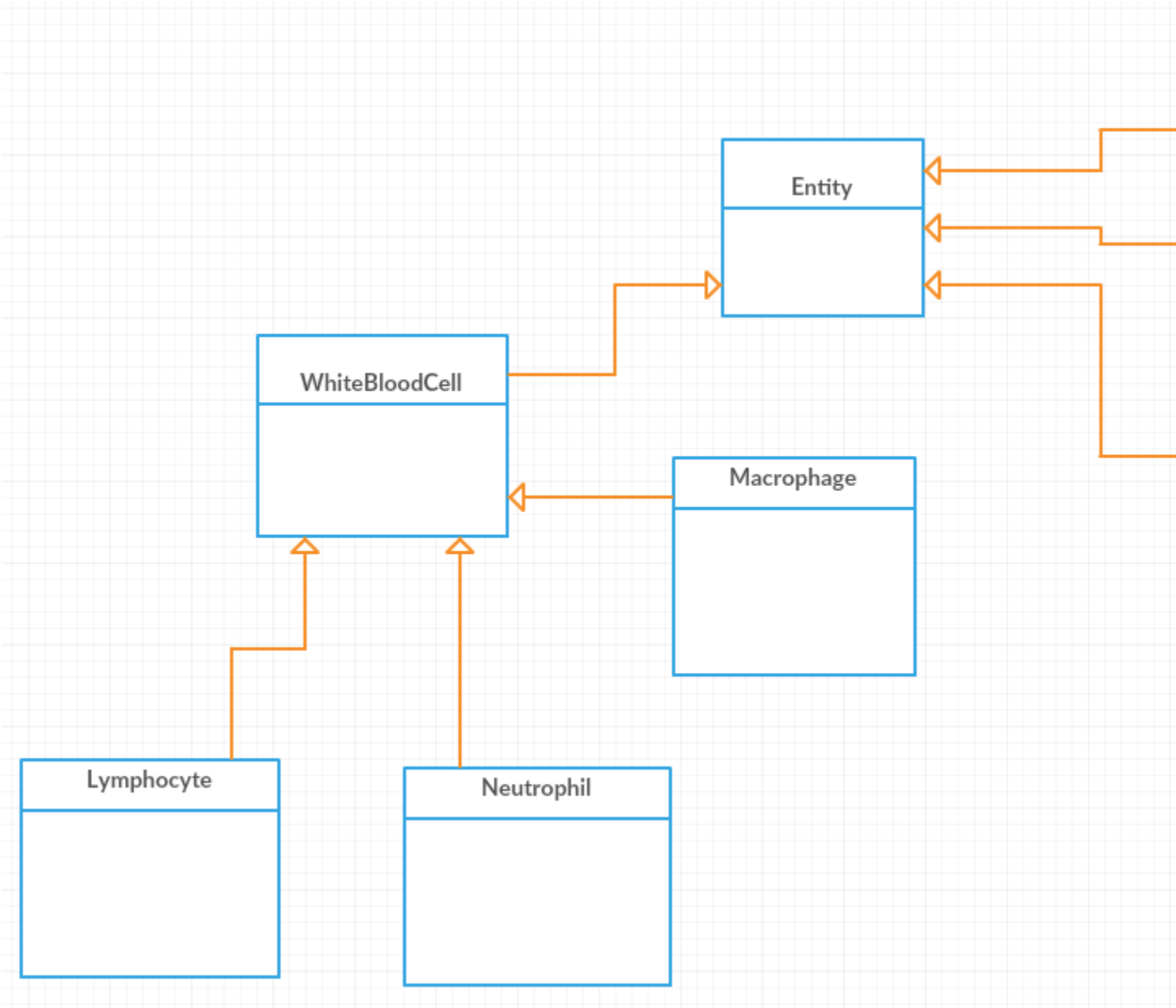
their scope, there are various articles and even books devoted to more sophisticated modeling of immune systen functionality. In this projects, we will use a few basic classes to explore relationships between various blood entities and their ultimate impact on patient health.

This assignment will ask you to write several classes, which we will detail below. Once you've written one or more classes, you can test them against our test cases. When all of your classes are completed, you can run them against a simulator we've written to show how these various entities could interact in a human's bloodstream.

You will have to do the following for this assignment:

- Create a project in Eclipse (or your IDE of choice) for this assignment
- Complete the `Entity.java` abstract class
- Complete the `Strength` enumeration
- Complete the `WhiteBloodCell` abstract class
- Complete the `Bacteria` class
- Complete the `Virus` class
- Complete the `Neutrophil` class
- Complete the `Lymphocyte` class
- Complete the `Macrophage` class
- Download the `Cytokine` class
- Download and run the `UnitTests.java` to check your work
- Download and run the `Simulator.java` to analyze your code

The diagram below shows the relationship between various classes:



Step 1: Write the following classes

```
abstract class Entity
```

The purpose of this basic class is to model behaviors of all the particles in the bloodstream. All of the other classes you'll write for this project will inherit from this base class. We've completed most of this class for you; below is a description of its elements, including the abstract method you'll define.	
ATTRIBUTES (please do not change their names)	
`private String DNA`	The DNA of an entity will be used to determine the type when two entities come into close contact in the bloodstream; some are friendly to one another, others are enemies!
`private Strength strength`	Assuming the entity is alive, how much power does it have to interact with the bloodstream? `Strength` is an enumeration.
`protected static Entity[] bloodstream`	This is a shared (static) attribute amongst all objects of this type, representing all the contents of the bloodstream. It is also protected, unlike the other private attributes.
METHODS (please do not change their names)	
`public Entity(Strength strength)`	This constructor will make the entity alive, and set its strength to the incoming argument.
getters/setters for `strength` and `DNA`	Use the *Source* option to automatically generate these in Eclipse.
`public void kill()`	This will set `strength` to `DEAD`.
`public abstract void touchNeighbor(Entity neighbor)`	This abstract method will be implemented by `Entity`'s child classes, to decide what happens when two entities touch in the bloodstream
`public static void setEntity(int size)`	This will set the size of the array of entities to the incoming argument.

`enum Strength`

The purpose of this enumeration is to code the four possible levels of power an entity has: `DEAD`, `LOW`, `MEDIUM` and `HIGH`.

abstract class WhiteBloodCell	
The purpose of this class is to model the defensive white blood cells in the blood stream. It extends the `Entity` class.	
ATTRIBUTES (please do not change their names)	
private String nucleusType	Unlike other blood cells, white blood cells all have a nucleus.
private Entity[] targets	This is an array of the types of entities this blood cell can neutralize.
METHODS (please do not change their names)	
public abstract void absorb(Entity in)	This abstract method will allow the white blood cell to absorb an invader.
public abstract void release()	This abstract method will allow the white blood cell to release something into the bloodstream.
public WhiteBloodCell(String nucleusType, Entity[] targets)	This constructor initializes all four attributes (including inherited ones). The `strength` will be set to high
public void touchNeighbor(Entity neighbor)	If the object is dead, this method will simply return without doing anything. Otherwise, this implements the abstract method of the parent class, by searching for matches of entities in the `targets` attribute against the incoming argument (by DNA). If a match is found, a battle is fought, using the following code which assigns a random number between 1 and 10 to the variable: `int battle = 1 + (int)(Math.random() * ((10 - 1) + 1));` If the `this` object's strength is `MEDIUM` and the battle's value is greater than five, or the strength is `HIGH` and the battle's value is greater than

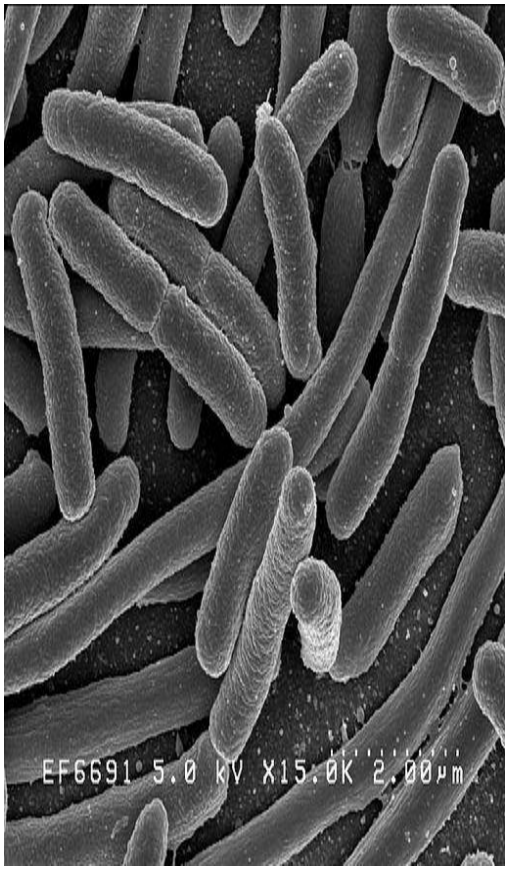
	one, the object will call its `absorb` method. Remember, a neighbor can also have a `null` value.

public void increaseStrength()	This method should be copied in to your file:
	`public void increaseStrength(){`
	`  switch (getStrength()){`
	`  case LOW:`
	`  setStrength(Strength.MEDIUM);`
	`  break;`
	`  case MEDIUM:`
	`  setStrength(Strength.HIGH);`
	`  break;`
	`  }`
	`}`

public void reduceStrength()	This method should be copied in to your file:
	`public void reduceStrength(){`
	`  switch (getStrength()){`
	`  case LOW:`
	`  setStrength(Strength.DEAD);`
	`  break;`
	`  case MEDIUM:`
	`  setStrength(Strength.LOW);`
	`  break;`
	`  case HIGH:`
	`  setStrength(Strength.MEDIUM);`
	`  break;`
	`  }`
	`}`

class Bacteria

The purpose of this class is to model bacteria in the bloodstream. It extends the `Entity` class.



METHODS (please do not change their names)

`public Bacteria(Strength strength)`	This constructor initializes both attributes (including inherited ones).
`public void touchNeighbor(Entity neighbor)`	This implements the abstract method of the parent class, to simulate bacteria multiplying. It will search the parent class' list of entities for the first `null` entry, and insert a new bacteria with `HIGH` strength into that location. On the other hand, if the object is dead, this method will simply return without doing anything.

`class Virus`

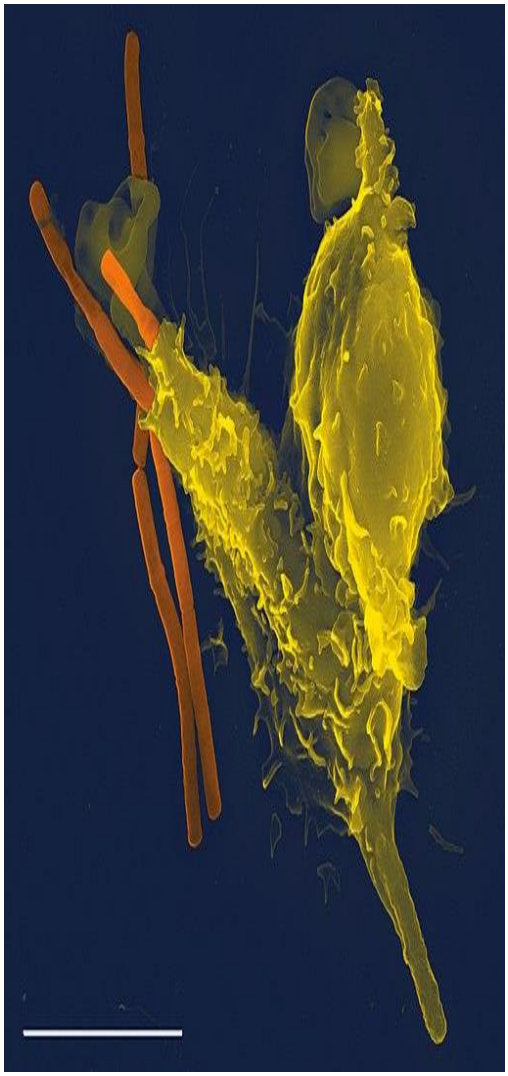
The purpose of this class is to model viruses in the bloodstream. It extends the `Entity` class.

METHODS (please do not change their names)

`public Virus(Strength strength)`	This constructor initializes both attributes (including inherited ones).
`public void touchNeighbor(Entity neighbor)`	This implements the abstract method of the parent class, to simulate viruses taking over a host cell. If the neighbor's DNA is a `neutrophil`, it will change the neighbor's DNA to a `virus` (that is, the object's type in Java will still be a `Neutrophil`, but its `DNA` attribute will now store the string `virus`). On the other hand, if the object is dead, this method will simply return without doing anything.

`class Neutrophil`

The purpose of this class is to model the invader-ingesting neutrophils in the bloodstream. It extends the `WhiteBloodCell` class.



ATTRIBUTES (please do not change their names)	
`private boolean sniffCytokines`	If this attribute is set, the cell can react to cytokines in the bloodstream.

METHODS (please do not change their names)	
`public Neutrophil(String nucleusType, Entity[] targets, boolean sniffCytokines)`	This constructor sets all attributes of the object (including inherited ones). The `DNA` type of the cell is set to `neutrophil`.

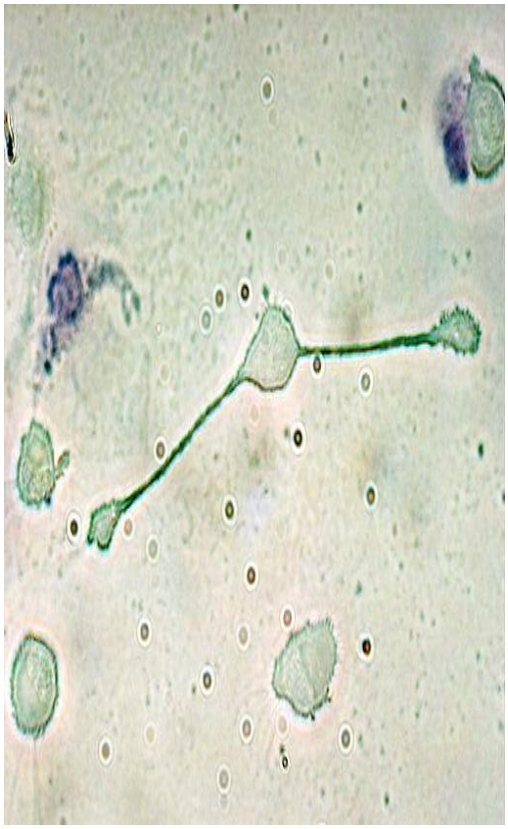
public void absorb(Entity in)	This overrides the parent's method by killing the incoming entity, reducing the cell's strength, and releasing into the bloodstream.
public void release()	This method will release three new `Cytokine`s into the `bloodstream` in the first `null` entries it finds, unless the object has been infected by a virus, as reflected in its DNA. In such a case, it will similarly release three new high-strength `Virus` particles into the `bloodstream` instead.
public void touchNeighbor(Entity neighbor)	This implements the abstract method of the parent class, to simulate a neutrophil sniffing for cytokines. It will first call the parent's `touchNeighbor` method, to chomp away at any existing targets. If it's set to sniff for cytokines, it will then see if the neighbor is a `cytokine`, in which case it will increase its own strength. On the other hand, if the object is dead, this method will simply return without doing anything.

class Lymphocyte

The purpose of this class is to model the battling lymphocytes in the bloodstream; these cells are short-lived and die off quickly. It extends the `WhiteBloodCell` class.

METHODS (please do not change their names)

`public Lymphocyte(String nucleusType, Entity[] targets)`	This constructor sets all attributes of the object (including inherited ones). The `DNA` type of the cell is set to `lymphocyte`.
`public void absorb(Entity in)`	This implements the abstract method of the parent class, and will kill the incoming entity and then call the `release` method.
`public void release()`	This implements the abstract method of the parent class, and will kill the current object with a ten percent chance every time it is called. You should google for a way to calculate a ten percent chance.

class Macrophage	
The purpose of this class is to model the invader-ingesting macrophages in the bloodstream; when these cells have ingested enough invaders, they eventually die. It extends the `WhiteBloodCell` class. 	
ATTRIBUTES (please do not change their names)	
`private int count`	This attribute keeps track of how many invaders have been ingested.
METHODS (please do not change their names)	
`public Macrophage(String nucleusType, Entity[] targets)`	This constructor sets all attributes of the object (including inherited ones). The `DNA` type of the cell is set to `macrophage`.
`public void absorb(Entity in)`	This implements the abstract method of the parent class, and will kill the entity, increase the count, and call `release()`.
`public void release()`	This implements the abstract method of the parent class, and will kill the current object after 100 invaders have been absorbed.

public void touchNeighbor(Entity neighbor)	This implements the abstract method of the parent class. If the object is dead, this method will simply return without doing anything. It will first call the parent's `touchNeighbor` method, to chomp away at any existing targets, and will then see if the neighbor is a `neutrophil`, in which case it will absorb this neighbor if the neighbor is `LOW` on strength.

class Cytokine	
The purpose of this class is to model the chemicals various blood cells can use to communicate. It extends the `Entity` class.	
METHODS (please do not change their names)	
public Cytokine()	This constructor sets the strength to `DEAD` (as it is not a cell). The `DNA` type of the cell is set to `cytokine`.
public void touchNeighbor(Entity neighbor)	This cytokine will kill itself if its neighbor is a neutrophil (to model being sniffed by such a cell).

Step 2: Testing Your Code For Functionality and Elegance

Please log in again to view these unit tests and simulator code.

The unit tests will, in addition to correctness, measure the following elegance metrics:

Coding style and readability	Make sure you use descriptive variables names, proper indentation, etc.
Class, attribute, and method documentation	Make sure you comment the purpose of each of these items.

Implement abstract methods correctly	• `WhiteBloodCell` should implement the parent's `touchNeighbor` method. • `WhiteBloodCell`'s `touchNeighbor` method should call its `absorb` method. • `absorb` and `release` should be abstract methods in `WhiteBloodCell`.
Override methods correctly	• `Neutrophil` should override the parent's `touchNeighbor` method. • `Macrophage` should override the parent's `touchNeighbor` method.
Proper use of inheritance for code reuse	• `Neutrophil`'s `touchNeighbor` method should call the parent method. • `Macrophage`'s `touchNeighbor` method should call the parent method. • Avoid attribute shadowing in any of the child classes.
Inheritance correctness and safety	• All child classes appropriately use the parent class' constructors. • The `@Override` tag is used in all the appropriate places. • All appropriate attributes are private.

Step 3: Running your code as a simulation

Now that you've successfully completed and debugged your code, you can use your classes to simulate what might happen in a healthy and a sick patient by running `Simulator.java`. Each simulation will have one thousand runs where will reshuffle their bloodstream array, and have each entity in the bloodstream touch its neighbor. You can view how the percentage of live and dead pathogens and white blood cells changes over time as these battles are fought; remember, that because there is random chance in our model, the final percentages may vary, sometimes significantly, though there will be overall trends between the healthy and sick patients.

Feel free to play around with the simulator code, for example:

- You can change the various entities each bloodstream for the healthy and sick person starts out with in their respective `healthyModel` and `sickModel` methods.
- You can update the `printStats` method to generate more information.

- You can modify how many times the blood is shuffled and entities touch their neighbors in their respective `healthyModel` and `sickModel` methods.

Step 4: Preparing for Assessment4

Create a copy of all of your files in a separate directory (preferably a new project in Eclipse). Then, modify those files to get them to pass the unit tests of Sample Assessment 4.