Chloe/Josh Brown
OOAD

Project 1

1. Provide definitions for each of the following terms. (Remember to cite sources, even if the source is the textbook. Wikipedia is not usable as a primary source.)

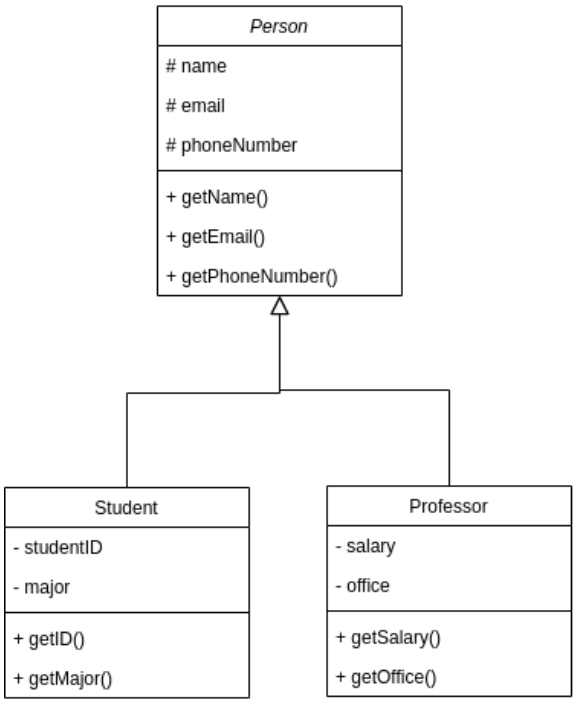
- abstraction
- encapsulation
- polymorphism
- cohesion
- coupling
- identity

Besides the definition, discuss how the term applies to the object-oriented notion of a class. Provide examples of both good and bad uses of these terms in the design of a class or a set of classes (can be code or a text/graphic example)

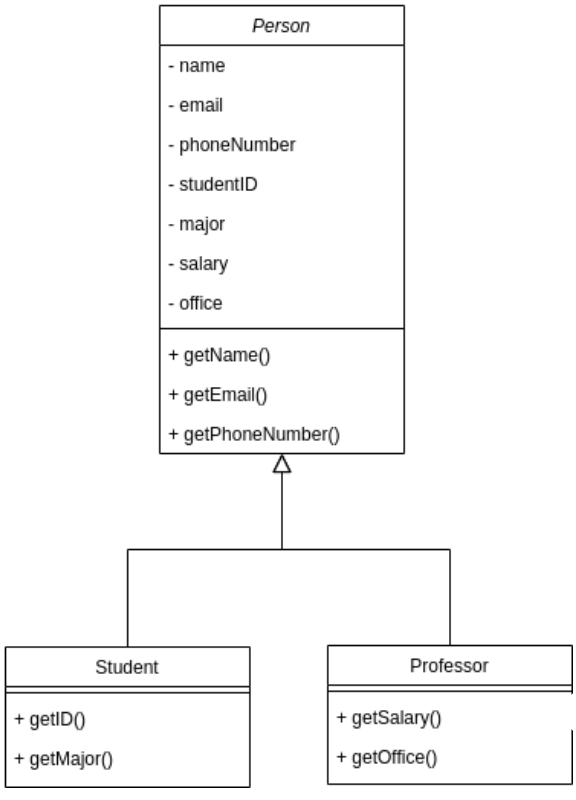
Abstraction

Abstraction is separating the complexity of an object into a more generalized form. I think of abstraction like how scientists define different species, each species has an ancestor that it stemmed from until you get to the most basic ancestor. I also think of abstraction as making a template for any objects that need a certain set of base functionalities or variables that I know other objects will all use.

Good



Bad



Encapsulation

Encapsulation is a way of hiding data and/or implementation within a class. Encapsulation can be used to restrict access to variables within a class to prevent outside classes from modifying them directly, we do this by making the variable “private.” We can then control access to the variable through the use of getters and setters, public functions that can be called by outside classes. We can also hide implementation by only providing a public method and let that object to all the work behind the scenes.

Bad example. Let’s say we have a timer class

```
public class Timer {
  public int seconds = 0;
  start() {
     // code that runs timer and increments seconds every second
  }
}
```

If a new Timer is instantiated and the start method called the timer will start counting up every second. But this allows the seconds variable to be modified outside of the class. Like this

```
public static main() {
  Timer t = new Timer();
  print(t.seconds); // 0
  t.start();
  sleep(3000); // sleep for 3 seconds
  print(t.seconds); // 3
  t.seconds = 10;
  sleep(3000); // sleep for 3 seconds
  print(t.seconds); // 13
}
```

The last print statement prints out 13 despite the timer only actually running for 6 seconds. To fix this with good encapsulation we can change Timer class to this

```
public class Timer {
  private int seconds = 0;
  public getSeconds() {
    return seconds;
  }
  start() {
     // code that runs timer and increments seconds every second
  }
}
public static main() {
  Timer t = new Timer();
  print(t.getSeconds()); // 0
  t.start();
  sleep(3000); // sleep for 3 seconds
```

```
  print(t.getSeconds()); // 3
  // t.seconds = 10; this would throw an error because the variable is
  // private
  sleep(3000); // sleep for 3 seconds
  print(t.getSeconds()); // 6
}
```
Now the seconds variable can't be modified outside of the Timer object itself.

Polymorphism

Polymorphism, at least my understanding of it, is when you instantiate a subclass as the type of its parent, like `Animal d = new Dog()`. Polymorphism is something I understand how to use better than I can describe with words. From lecture 4 slide 51, polymorphism is defined as "*in OO A&D it means that we can treat objects as if they were instances of an abstract class but get the behavior that is required for their specific subclass.*" My understanding of it is that we can instantiate several objects as a generic parent but we can ask it to do the same thing with different outcomes depending on the subclass.

Good example, Animal objects that are asked to speak.

```
public static main() {
  List<Animal> animals = new List<Animal>();
  animals.add(new Dog());
  animals.add(new Cat());
  animals.add(new Parrot());
  for (Animal a : animals) {
    a.speak();
  }
}
```
This would print:
```
Bark
Meow
Squawk
```
This is not doable without polymorphism and instead speak() would have to be called for each object individually.

```
public static main() {
  Dog d = new Dog();
  Cat c = new Cat();
  Parrot p = new Parrot();
  d.speak(); // Bark
  c.speak(); // Meow
  p.speak(); // Squawk
}
```

Cohesion

Before this class I didn't know what cohesion was in terms of object oriented programming. From lecture 4 slide 27, cohesion is defined as "how closely the operations in a routine are related" and "we want this method to do just one thing." I like the second description of cohesion and in my own words I would put cohesion as being an object has one job so that other objects don't have to do that job. A good example of cohesion (strong cohesion) would be a User object for some login system. That User class would have basic get and set methods for that users data (e.g. `getName(), getAddress(), getID()`), and that is all it should do. I bad use of cohesion (weak cohesion) is if we added a whole bunch of implementation that it doesn't need like `updateUserDatabase(), login(), payBill(), goToBank(),` all of which could be implemented in different classes so that the User class only has the one job of holding the users data.

Coupling

Before this class I also didn't know what coupling was in terms of object oriented programming. From lecture 4 slide 28, coupling is defined as "the strength of a connection between two routines." I understand this as "how dependant is this object to other objects," or simply how many dependencies an object has. A good example of coupling (strong coupling) would be any class that needs to import little to no additional classes in order to function properly. A bad example would be a class that relies on several other classes in order to function properly, a single change to one of those dependencies could lead to breaking that entire class.

Identity

Identity is fairly simple, it's how we identify objects from one another and how we can compare them to see if they are equal or not. An object without an identifier of some kind becomes hard to distinguish from each other. Say we have a Train with passengers that don't have any kind of identifier and we want to count how many people are on the train. We simply would count how many of the Passenger objects the Train object holds. What if we now want to count how many people are in a specific car on the train, we can't, there is no way of identifying how many people are in a specific car of the train. The only way to solve this would be to add additional variables to the Train class to keep track of the passengers for each car. The way to do this with identity would be to give each Passenger a carNumber variable to identify which car that passenger belongs to, then we can identify which car a passenger is in based on their carNumber and count the number of passengers for that car number.

2. A company has asked us to design a payroll system that will pay employees for the work they perform each month. Using a level of abstraction, similar to that shown in slide 14 of lecture 4, develop a design for this system using the functional decomposition approach. You can assume the existence of a database that contains all of the information you need on your employees. For your answer, first describe the functional decomposition approach, discuss what assumptions you are making concerning this problem, and then present your design.

First identify the overall problem that needs solving, the company wants a system that pays employees based on their work each month. Now to break that into individual problems that are smaller and more manageable. We want to pay all the employees, so we need to pay each employee individually. To pay an employee we need to know how much to pay them, how do we calculate how much to pay them? Let's assume an employee's salary is calculated on the number of hours they worked and how much they get paid per hour. So in order to calculate an employee's paycheck we would simply multiply the number of hours worked by the amount they make per hour. Now we know how much an employee needs to get paid, now how do we pay them? Does that employee have direct deposit or will they get a physical check? We should have a specific function that will handle how an employee receives their money and sends payment accordingly.

Assuming database for employees:
id | name | wage | hoursWorked | paymentMethod | bankAccountNumber | bankRoutingNumber | address

Pseudo code design:

```
payEmployees() {
  // get all employee data from database
  employeeList = getEmployees()

  for employee in employeeList {
    // calculate paycheck for a given employee
    paycheck = calculatePaycheck(employee)

    // pay that employee
    payEmployee(employee, paycheck)
  }
}

getEmployees() {
  // connect to database and return all employee data
}

calculatePaycheck(employee) {
  wage = employee[wage]
  hoursWorked = employee[hoursWorked]
  return wage * hoursWorked
}

payEmployee(employee, paycheck) {
  if(employee[paymentMethod] = direct deposit) {
    // get employee bank account information
```

```
    bankAccountNumber = employee[bankAccountNumber]
    bankRoutingNumber = employee[bankRoutingNumber]

    depositPaycheck(bankAccountNumber, bankRoutingNumber, paycheck)
  } else {
    address = employee[address]
    name = employee[name]
    sendPaycheck(name, address, paycheck)
  }
}

// depositing a paycheck requires the account number and the amount
paid
depositPaycheck(bankAccountNumber, bankRoutingNumber, amount) {
  // do whatever sends the paycheck to the bank
}

// a physical paycheck needs a name, an address, and the amount paid
sendPaycheck(name, address, amount) {
  // do whatever sends a physical paycheck to employee
}
```

3. Now develop a design for the payroll system using the object-oriented approach, keeping in mind the points discussed on slides 30-32 of lecture 4. Identify the classes you would include in your design and their responsibilities. (As before, you can assume the existence of a database and that you'll be able to create objects based on the information stored in that database.) Then, identify what objects you would instantiate and in what order and how they would work together to fulfill the responsibilities associated with the payroll system.

We need classes that each need to do tasks based on what you think that class should be able to do. We need a Database class that interacts with the database, an Employee object that holds all the relevant information for the employee, a Payroll class that holds all the employees, a Paycheck class that will handle how to pay the employee with the subclasses DirectDeposit and PhysicalCheck, and finally a BankAccount class to hold the bank account information

This object interacts with the database to retrieve a list of json data.

Database

```
+ getEmployees(): json
```

```
getEmployees(){
  jsonData = query("select * from employees")
  return jsonData
}
```

getEmployees()
returns list of json data objects.

Employee

```
+ name: string
+ wage: int
+ bankAccount: BankAccount
+ address: string
+ hoursWorked: int
+ paymentMethod: string
+ paycheck: Paycheck
+ calculatePaycheck(): void
```

```
Employee(json) {
  this.name = json[name]
  this.wage = json[wage]
  this.hoursWorked = json[hoursWorked]
  this.paymentMethod = json[paymentMethod]
  // this may instantiate the BankAccount object this null values
  this.bankAccount = new BankAccount(json[bankAccountNumber], json[bankRoutingNumber])
}
calculatePaycheck() {
  amount = wage * hoursWorked
  if(this.paymentMethod == "direct deposit"){
    this.paycheck = new DirectDeposit(this.bankAccount, amount)
  } else {
    this.paycheck = new PhysicalCheck(this.name, this.address, amount)
  }
}
```

Employee(json)
The constructor sets all the necessary variables given the json data.
calculatePaycheck()
Calculates paycheck amount and set paycheck variable to DirectDeposit or PhysicalCheck depending on paymentMethod.

This object holds a list of Employee objects and is responsible for paying all of the employees.

Payroll

```
+ employeeList: List<Employee>
+ payEmployees()
```

```
Payroll(jsonData) {
 employeeList = new List<Employee>
```

```
  for json in jsonList {
    employeeList.push( new Employee(employeeJSON) )
  }
  this.employeeList = employeeList
}
payEmployees(){
  for employee in this.employeeList {
    employee.calculatePaycheck()
    employee.paycheck.send()
  }
}
```

Payroll(jsonData)

The constructor takes a json list and populates the employeeList variable with Employee objects.

payEmployees()

Loops through the list of employees and calls the calculatePaycheck for each employee and then sends the paycheck.

This object holds how much an employee will be paid.

Paycheck
`# amount: int` `+ send(): void [abstract]`

`Paycheck(amount) {`
`  this.amount = amount`
`}`

send()

This is an abstract method so the DirectDeposit and PhysicalPaycheck classes must implement it.

This is a subclass of Paycheck and is responsible for depositing the paycheck directly into the employees bank account when the send function is called.

DirectDeposit (extends Paycheck)
`- bankAccount: BankAccount` `+ send(): void`

`DirectDeposit(bankAccount, amount) {`
`  super(amount)`
`  this.bankAccount = bankAccount`
`}`
`send(){`
`  // send a direct deposit using the this.bankAccount objects information`
`}`

send()

This will implement the Paycheck send() function to specifically send a direct deposit using the this.bankAccount objects information.

This is a subclass of Paycheck that is responsible for sending a physical paycheck when the send function is called.

PhysicalCheck (extends Paycheck)
- address: string - addressTo: string + send(): void

PhysicalCheck(addressTo, address, amount) {
  super(amount)
  this.addressTo = addressTo
  this.address = address
}
send(){
  // send a physical check using the addressTo and address information
}

send()
This will implement the Paycheck send() function to specifically send a physical check using the addressTo and address information.

This is a simple object to hold the bank account information in one place.

BankAccount
+ bankAccountNumber + bankRoutingNumber

BankAccount(bankAccountNumber, bankRoutingNumber) {
  this.bankAccountNumber = bankAccountNumber
  this.bankRoutingNumber = bankRoutingNumber
}

This instantiates the Database object and retrieves the user data. Then, a Payroll object is created with the json data which automatically creates all the necessary objects. Finally, the payEmployees function is called which pays the employees.
Psydo code design:

```
main() {
  db = new Database()
  jsonData = db.getEmployees()
  payroll = new Payroll(jsonData)
  payroll.payEmployees()
}
```

4. Write a simple OO program in Java 8 or later that implements a Zoo full of Animals. The Animals in your Zoo are represented on slide 16 of Lecture 5. Create a class structure similar to slide 17 with at least three levels of inheritance (e.g. Animal, Feline, Cat). You may

decide how many of each animal live in your zoo, but there should be at least two instances of each subclass (like Cat). Individual animals should have individual names for their instances that start with the same letter of their subclass (e.g. Charlie and Chloe for Cat).

You will also need a class for a Zookeeper. The Zookeeper will have the following responsibilities – wake the animals, roll call the animals, feed the animals, exercise the animals, shut down the zoo. Each animal will have a method that responds to this (wakeup, make noise, eat, roam, sleep). You can decide when it's appropriate to use a method at a superclass or subclass level for each animal's behavior, but there should be some behaviors that are placed at each of the three inheritance levels. In at least one case (probably for the Cats) use a random number generation to select from alternative responses to animal actions (e.g. when you ask a cat to sleep, it may randomly sleep, hiss, or run around).

When the program runs, the Zookeeper should execute each of his responsibilities in order (display a string for each action he executes), and the animals should respond by displaying strings with their name, their type, and their response to the Zookeeper. It is likely you will have a main program that instantiates the objects before they begin to act. Your program should demonstrate polymorphism by asking your collection of animals to perform their tasks by referring to a collection of all animals at the Zoo when executing methods. Capture the text output of the final program in an out file called "dayatthezoo.out". Your submission must include your Java code, a README (described below), and this out file.