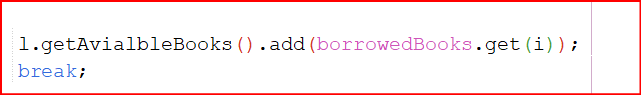
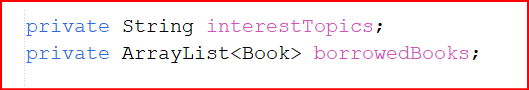
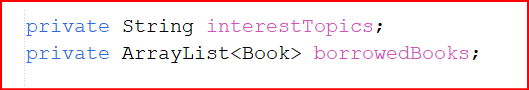
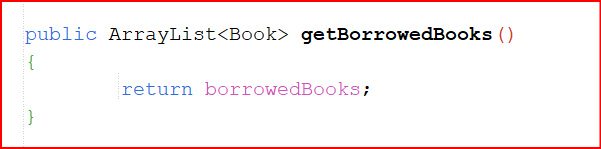
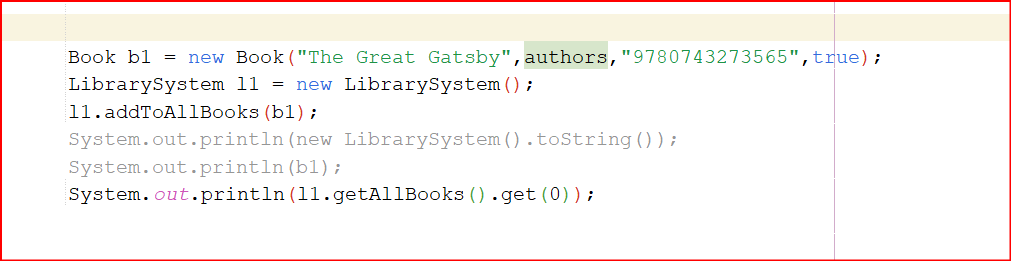
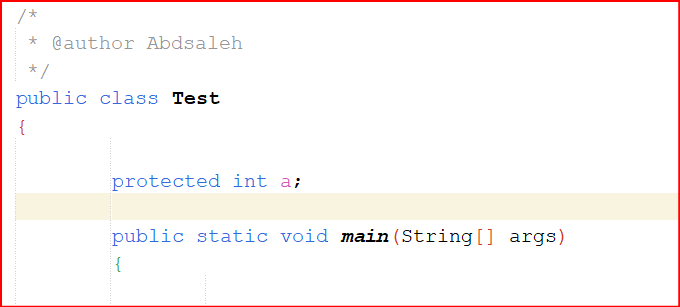
**Student Assessment Submission and Declaration**

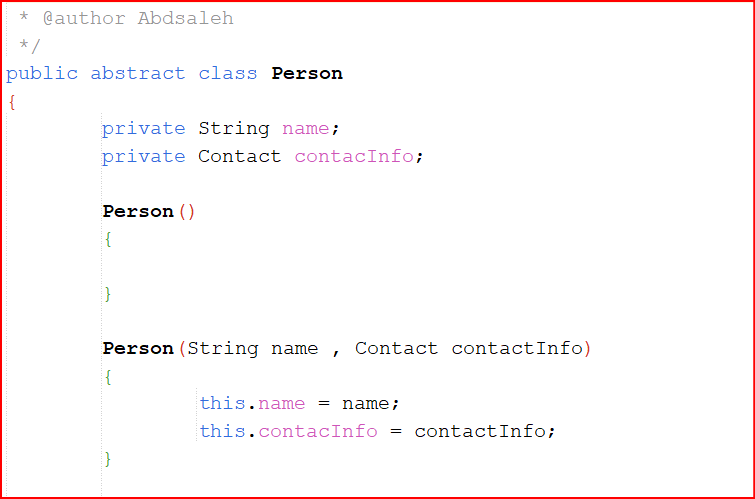
When submitting evidence for assessment, each student must sign a declaration confirming that the work is their own.

|  |  |  |  |
| --- | --- | --- | --- |
| Student name: Abdelrahman Saleh | | Assessor name:  **Abdelbaset Assaf, Hazem Al-Najjar** | |
| Issue date (1St Submission):  **30/03/2023** | Submission date (1St Submission):  **20/04/2023** | | Submitted on 20/04/2023: |
| In case of resubmission | | | |
| Issue date (2nd Submission): | Submission date (2nd Submission): | | Submitted on: |
| Programme: Higher National Diploma in computing – Software Engineering, Intelligent Systems | | | |
|  | | | |
| Assignment number and title: 1: OOP Concepts | | | |

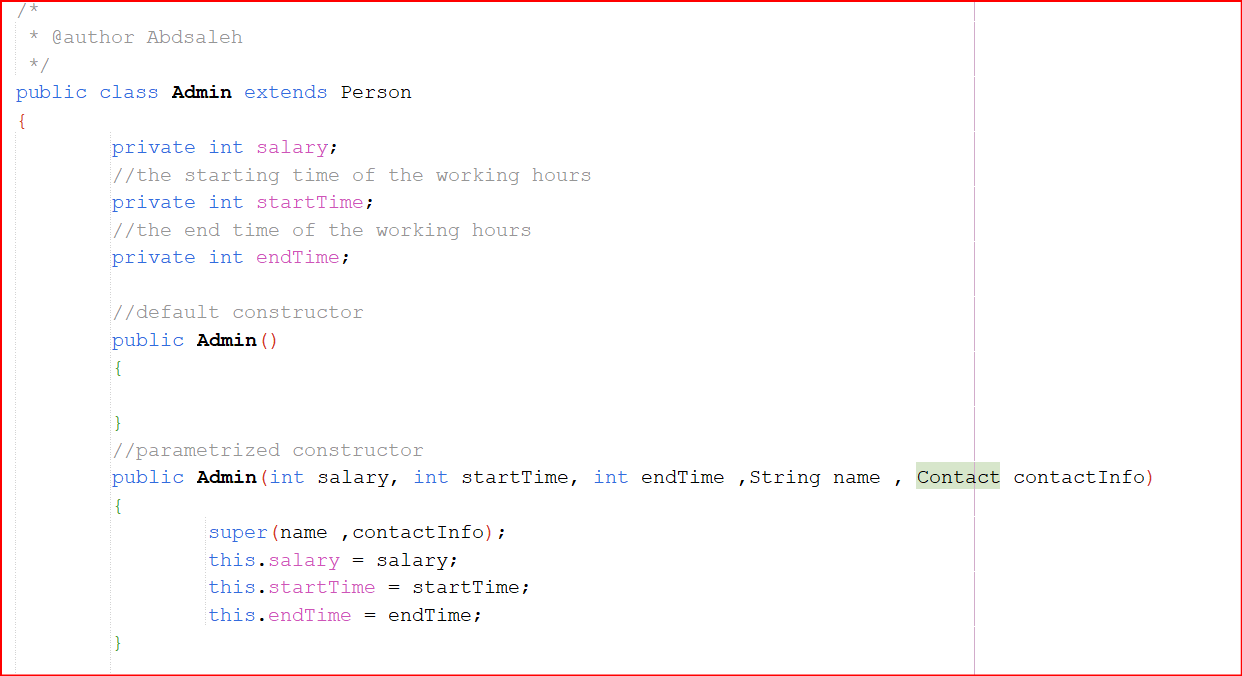
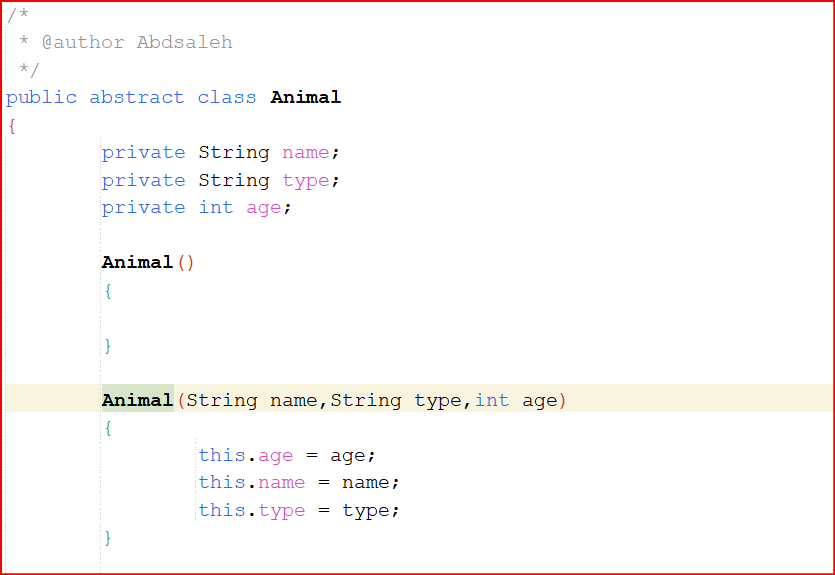
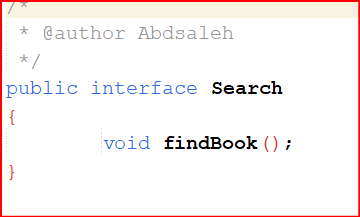
**Task.1)**

1. Access modifiers and Encapsulation:   
     
   Encapsulation: It is basically represent hiding the information from any foreign manipulation to ensure that the object’s data is not accessed or modified inappropriately. In other words we can say that the purpose of encapsulation is to protect the internal state of an object and provide controlled access to it through proper ways (like setters and getters), and it also involve the hiding of the implementation of a certain method so we can you use it without knowing how exactly it functions or what implementation it has.   
     
     
     
   For example here I used the (.add ) to add an object to that list so I used the method without really knowing how it works and what implementation it has so that is a part of encapsulation,   
   also by declaring a certain variable or method as private we use encapsulation and prevent other classes to access this method/variable directly and modify it inappropriately unless they use a proper way like using the setters and getters methods that we implemented in that class.  
     
   Access modifiers: In short they define the visibility and accessibility of attributes and methods within a class, We have 4 types of modifiers in java, which are:  
   1) Private: Using the private modifier is basically saying that the variable that we declared as private is not to be accessed via other classes directly even with an instance of this class, the only way to access it is using the methods in the class that has direct access to that variable.   
     
   2) Public: The public modifier refers that the member declared as public are accessible via all other classes using instantiation , and can be accessed directly and modified as needed without the need to call a certain method or use other ways to access that member of another class or other packages or even other projects so it has a broad scale of accessibility to almost all parts. Note that even if the subclass is located outside the same package it is still has access to the members is the original calss.  
     
   3) Protected: Protected members are accessible within the class itself and its subclasses in the same package, so they are not accessible from outside the same package or if its not a subclass of the class, so we can say that it is not accessible unless it is within the same package or by using inheritance.  
     
   4) Default: The default modifier is not really a specific modifier that we use to declare a variable, but rather it is the default state for the variable/method to be in if you didn’t specify the modifier for it, so by just declaring a variable by saying (int a) this automatically will be assigned the default modifier to indicate that this member is not accessible by other classes outside of the same package that that member is located in, even if it is a subclass of the original class it will still not have access to the members in the super class if it is located outside the same package.  
     
     
     
     
   Here is an example of one the access modifiers that we discussed earlier which is private that as I mentioned above strict the access to only the same class.  
   And here is another example of another access modifier which is the public modifier that gives access to this member throughout the entire code base inside and outside the same package.  
   And here I didn’t specify the modifier type for the b1 object so it automatically got the default modifier which gives access to it throughout the same package.  
   Here is an example of the protected modifier, here the variable ‘a’ is accessible by any part within the same package as it, and also it is accessible by the subclasses of it’s class even if they are outside the same package.
2. Composition: Simply It is using an instance of a class as an attribute in another class to work together and make things easier for us to use cause some languages such as java does not support multiple inheritance so we have to resort to some alternative method, some of them are the use of composition, So to give an example of it we can use the scenario that we have, in our scenario there is a person and every person has a contact info so in order to use that and make use of it to the fullest we use composition and create another class called contact for example and put in it all the attributes that we want our person class to composite, and we can be really specific as much as we want like having an email, address, phone number , social number or any other piece of information that we may find useful to in our case, after that we make an instance of the contact class within the person class and make use of the attributes in the contact class in the person class, so it like a collaboration between classes to achieve a goal, and we can think of the relationship between the two as “has a” relationship, in our scenario the person has a contact info. Text, email

   Description automatically generated

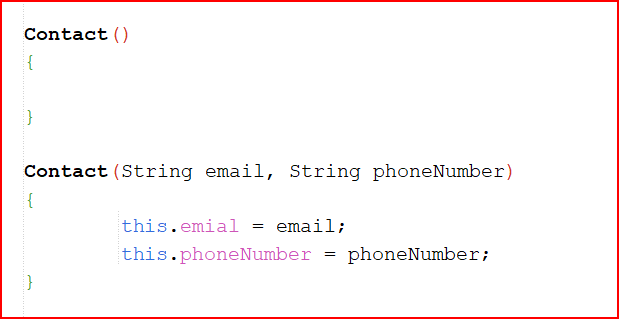


**Here is an example of it with the same classes.**

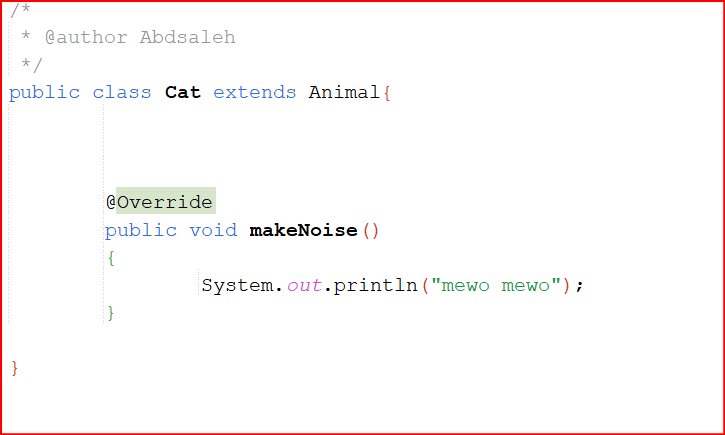
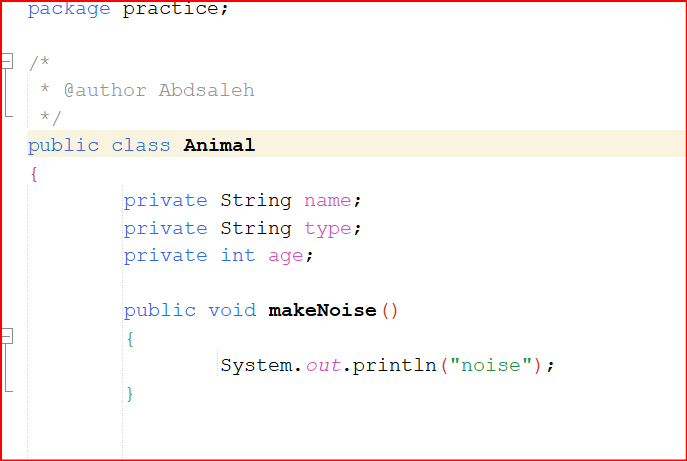
1. Inheritance: In simple terms inheritance is inheriting the properties and behaviour of another class, for example if we want to represent a cat, we can implement a class that represents it but we would have to do that for every other animal if we have multiple ones, so instead of doing it multiple times we can simply declare the main attributes that all animals have in common in one class and let all the other animals inherit those attributes and properties, so we get a hierarchy of classes where subclasses can inherit the properties of the superclass/parentclass, and also can override them to provide specialized behaviour. We can describe this as saying that the cat is an animal so the inheritance relationship is represented by “is-a” relationship that gives us some sort of a template for all the subclasses that inherit the superclass.  
     
   Here is an example of inheritance that I made, here i have the class Admin that extends the class Person so here the subclass is Admin and the superclass is Person, the Admin class inherits all the attributes that Person has so Admin is a person, so adding to the original attributes that the admin has we will add the attributes that preson has as you can see within the constructer we call the “super” keyword that indicate that we call the superclass which is in our case the person class, and pass its attributes to fill the arguments that we have to fill in the admin constructor.
2. Abstraction: In a way abstraction in classes is to prevent it to be instantiated, like if we have an animal class that is a super class so it does not make sense to make an instance of that class in its own so we have to make another class that extends it and inherit its attributes like a cat for example, so in simple terms we can’t instantiate an abstract class but we can instantiate a subclass of the abstract class, Regarding the abstract methods we can declare a method as abstract to force every subclass of the abstract class to implement that method so every implementation may have unique characteristics that is different from class to class, like for example if we have an animal class and we have three subclasses of that class which are cat ,dog and a parrot, and we have an abstract method in the animal class called makeNoise to indicate that every animal makes noises but every one of them makes noise in their own way or have a unique voice, so make this method abstract to force each subclass to implement the method and represent the voice of that animal like a cat meows and the dog barks and the parrot squeak for example so each subclass will say that this animal makes a certain unique voice. Note that when we declare a method as abstract the method does not have a body so we can implement it in the subclasses, and of course we can implement each method individually in each class but by declaring as an abstract in the superclass we are in a way enforce and organize each subclass that inherits the superclass by forcing it to have this method implemented in it, like in our previous example we enforced each subclass to have the makNoise method in them because each and every animal should be able to make noise some how and we have to represent that in our code and not let it be an optional thing.  
     
     
   Here is an example of using abstract in a super class, I used abstract here to prevent the instantiation of this class on its own because its not even logical to make an instance of an animal on its own cause have to specify what kind of animal it is so will be forced to just make other classes inherit its attributes and represent different kinds of animals.  
   Here is an example of a method being abstract but here I used an interface, the interface in considered to be abstract by default and all its method and variables are abstract by default also, so here every class that implements this interface is forced to implement this method in its class using their own unique implementation.

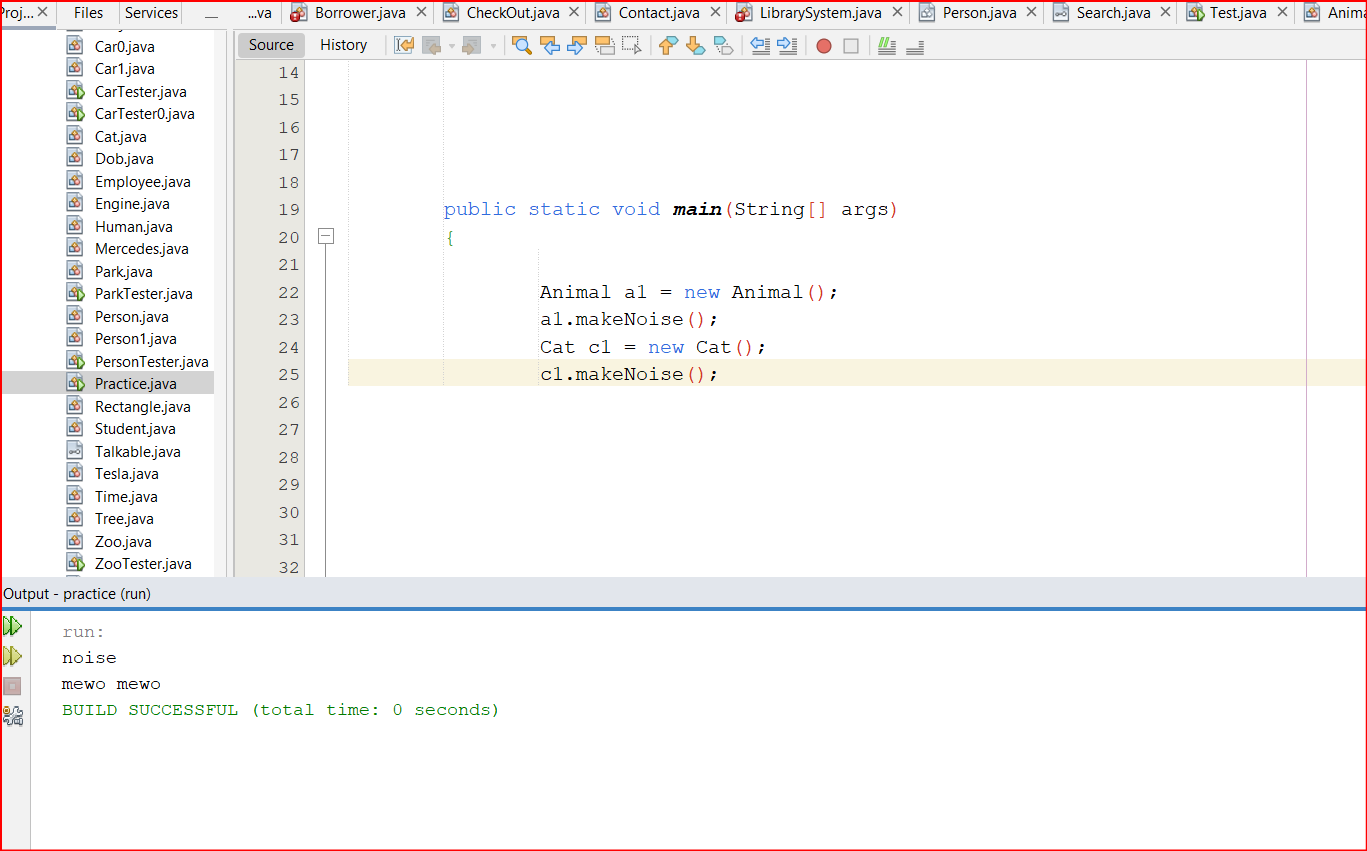
[(57) Abstract Classes and Methods in Java Explained in 7 Minutes - YouTube](https://www.youtube.com/watch?v=HvPlEJ3LHgE)   
More info on the topic of abstraction.

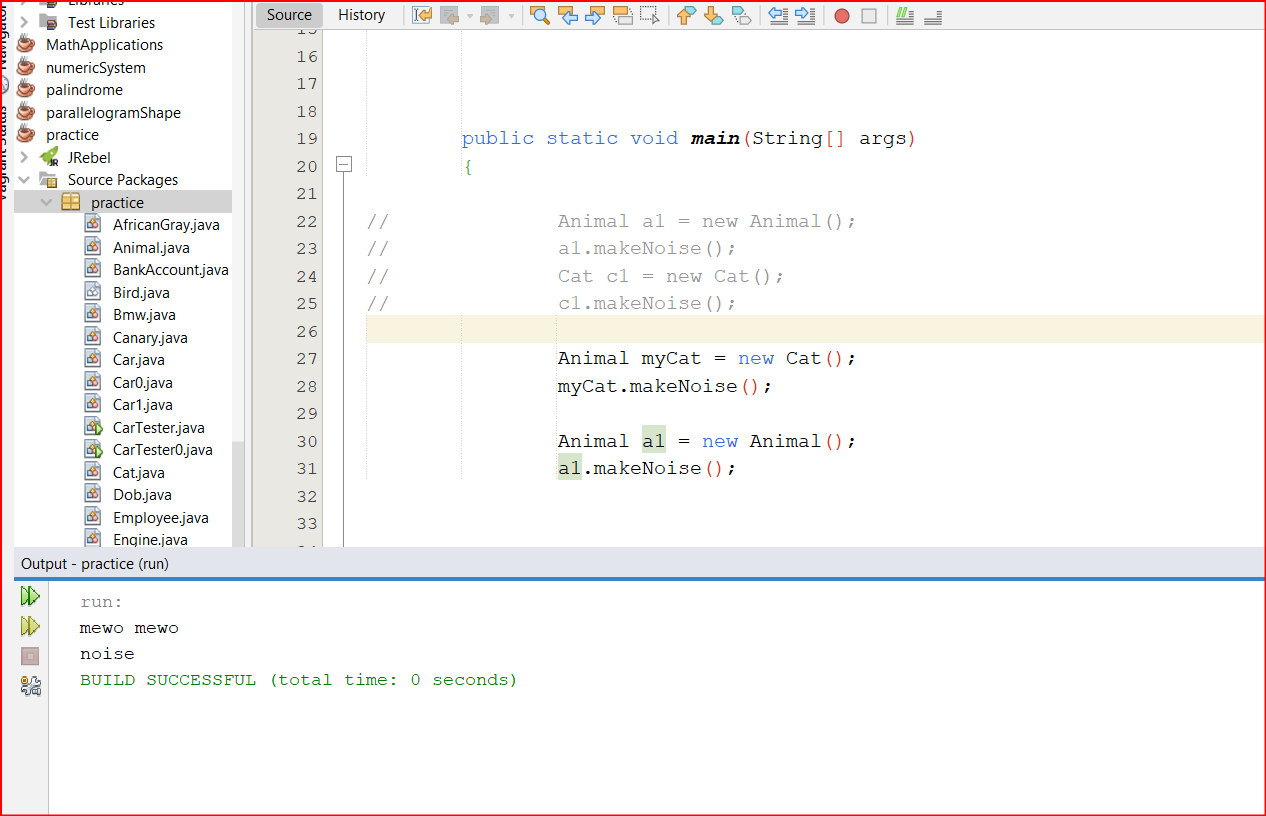
1. Polymorphism: Polymorphism is a feature that allows objects of different classes to be treated as if they were of the same type, enabling them to be used interchangeably, in other words or as i understood it allows objects of different classes to respond to the same method call regardless of their actual type, I will mention a couple of ways to use polymorphism to try and deliver my understanding about polymorphism.  
   we have,

Overloading: which is the existing of the same method with different parameters, it like having different versions of the same app but every version has an added feature for example or maybe a different target so in a way we can simply say that it is the act of having multiple methods with the same name, return type and modifiers but with different parameters for each one and every one of them can have its own implementation or have the same.

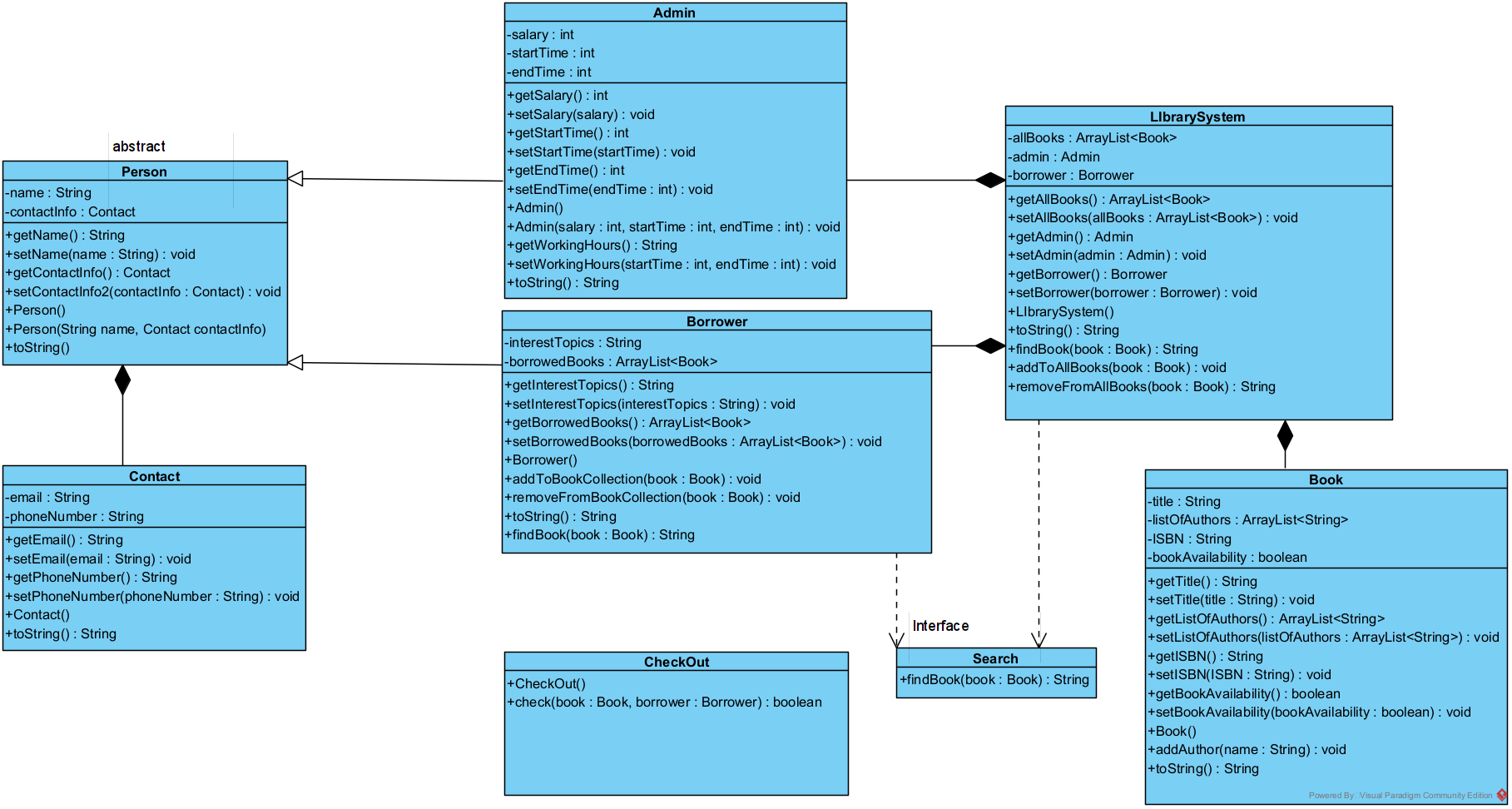
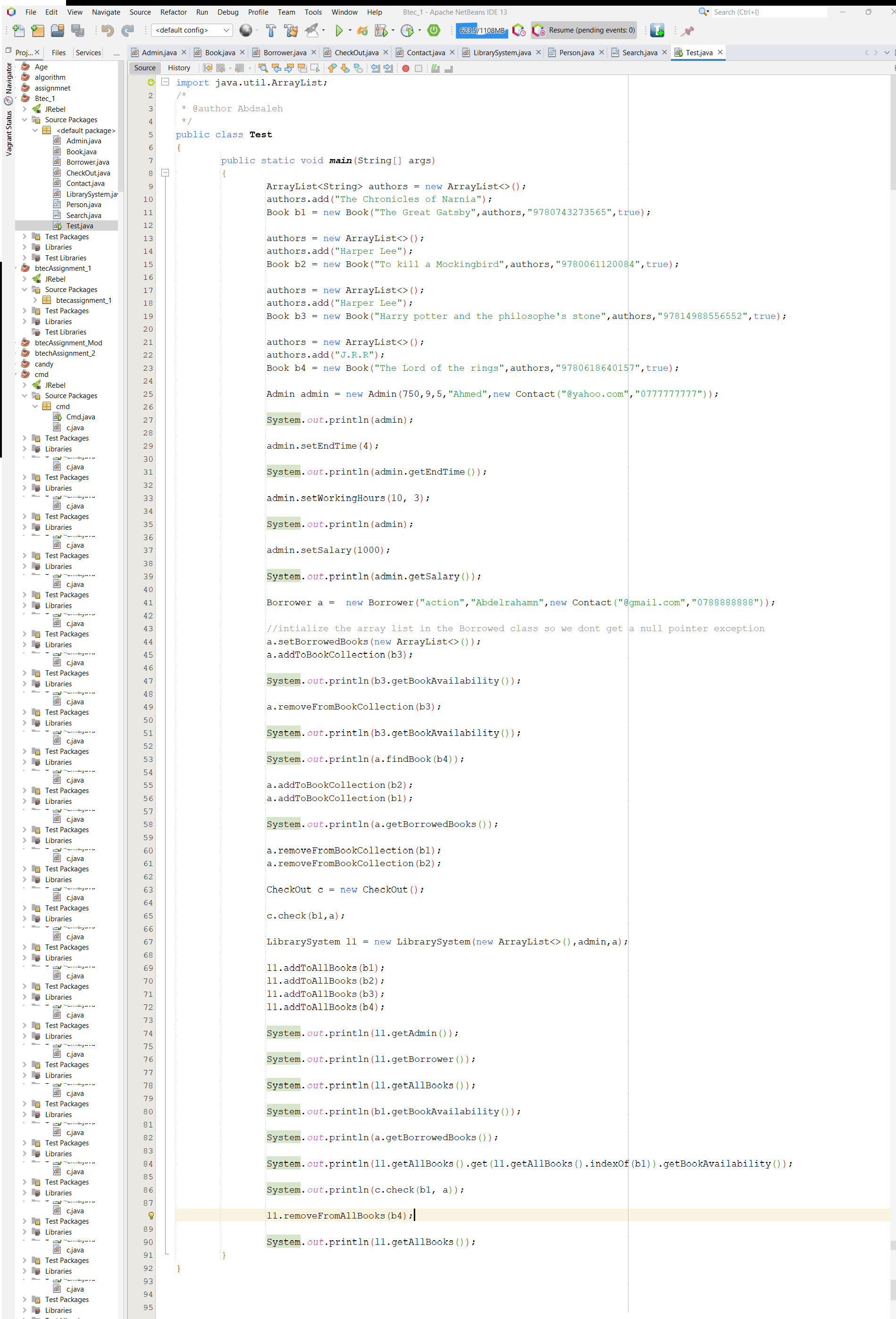
Here I have a constructor of a class named Contact and created two constructors, one of them is parametrized and the other is not parametrized so this is an example of using overloading in polymorphism.

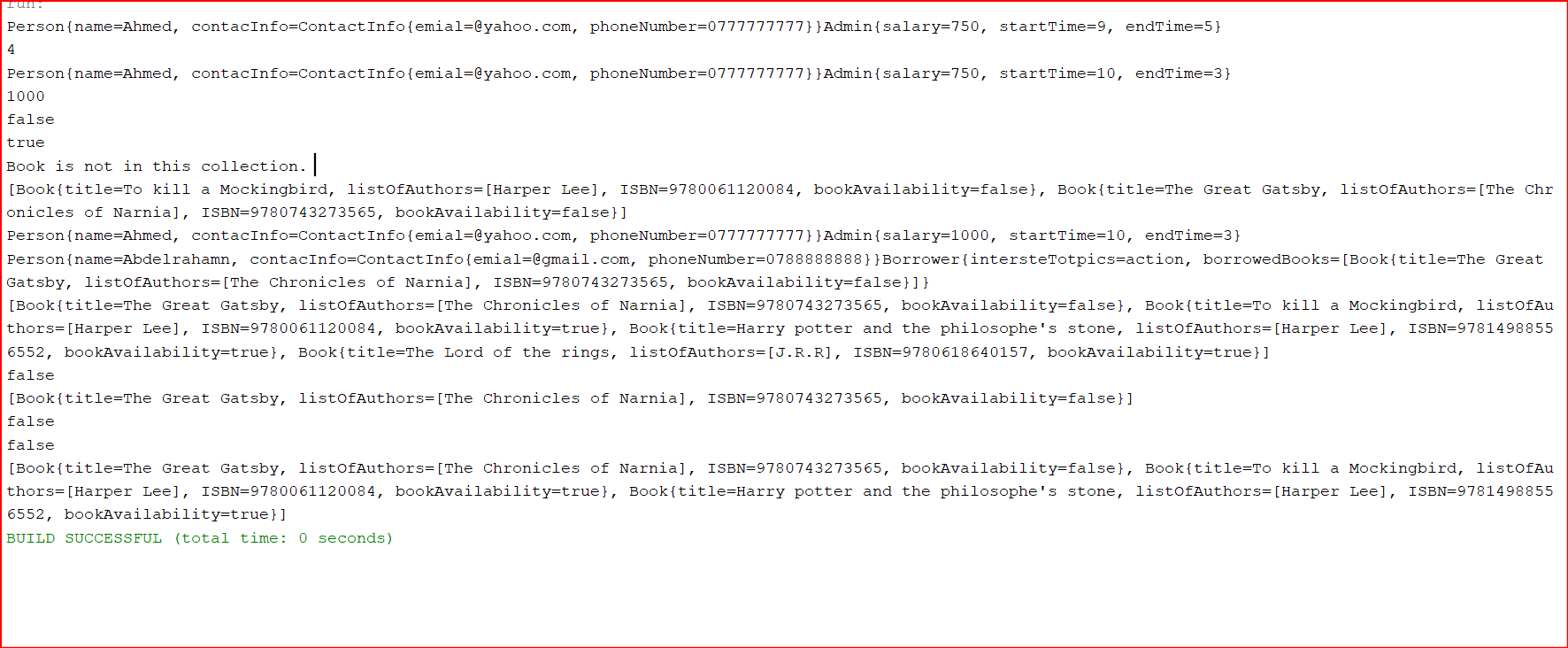
Overriding: Here the simple version of it is when we have a subclass of another class(Super class) and the super class has a method in it , normally the subclass will be able to access that method and use it at will but the subclass will be forced to use it original implementation in the super class, so here using overriding feature it can implement the method in it’s own body and write the needed implementation that he want to specify, but the subclass will be forced to write the method with the same name, access modifier and return type as it is in the super class. For further explanation we will have a simple example, lets say that we have a super class called Animal and we have two subclasses that inherits the Animal class called Dog and Cat, and in the super class we have a method called talk or makeNoise and we have a basic implementation for that method in the super class that just say “sound” but we don’t want the dog for example to make that sound, so we can rewrite that method in the dog class and have it say “hof hof “ for example so here we made a method in the subclass that overridden the original method in the super class. 

  
Here I created almost the same example, I first created the super class called Animal then implemented a method in it called makeNoise that just prints “noise” then created a Cat class and made it extends Animal so it became a subclass of Animal, then instead of just let Cat have access to the original makeNoise method I implemented it in the Cat class to customize it and make the cat make a different sound, in the implementation I let it print “meow mewo” then I made an object of the two classes then called each method through its own object and each one printed a different thing due to them having different implementation.

polymorphic variables: They are variables that can hold references to objects of different types, but they are declared with a common type, let us understand it through our previous explanation about overriding when we had an Animal super class and a Cat sub class and each of them had a different implementation of the same method called makeNoise, when we want to make an instance of them we can declare an object of type Animal(super class) then initialize it with Cat , so we basically saving the new Cat instance in an object of type Animal and that is simply the polymorphic variables, and by doing so when we call the makeNoise of that object that we just initialized it will print the implementation that we have in the Cat class which says “meow meow” and when we make an object of Animal and initialize it with an instance of Animal and call the makeNoise method through that object it well print “noise” according to its implementation in the Animal class.

**Here is an example of what I just say. Note that I used the same classes from the previous example with the same implementation (you can see them above for reference).**

**Task.2)Task.3)**

**Here is the Test class that I made to simulate the process of booking a book through the library system that we built earlier, The first step I took is to declare and initialize objects of books so we can have multiple books in our system, but before that I had to declare an arrayList of Strings to store the author/s names in it and pass that arrayList in the Book constructor and that what I did I have the authors arrayList declared and initialized then added the authors of the book that I want to create an object of to that arrayList and passed the other information that I need to call the Book constructor, which are the title of the book then the list of authors of that book then the ISBN number and the status of the book if it is available or not, I made 4 variables each of type book named b1,b2,b3 and b4 respectively, after that I made an admin and passed its parameters as well, the admin parameters are first its salary, starting time of work, end time of work , name, and lastly we have a composition between the admin class and the contact class which have the phone number and email variables , so we called the contact constructor in the admin constructor and passed the values the current admin, All of this we do to create a library system so we must have first books then an admin then borrowers and after that we put them all together in the librarySystem class, so I went through the process in order but I had to do some checking on the methods that we wrote in the admin class so I made some changes to the original data I passed within the constructor and printed out the results to see if the changes were successful , like setting a new salary or editing the working hours, after that I created a borrower and passed it parameters to it then made some interactions like if this borrower wanted to borrow a book then will it be added to his collection? Or will the book status get changed? So I used some of the method that I wrote in the Borrower class and tested them and made sure that they behave as intended, then I used the CheckOut class and made an interaction through this class and printed the results and they also works fine, then patched every thing together and made the system and passed an arrayList and an admin and a borrower as parameters and added the books to the arrayList and printed out some information through the system object called l1 I retrieved the admin information and the borrower information and every info war correct with no misinformation as you can see in the picture below I screenshotted the resulted data for every command that I wrote as proof of working system. **

**Task.4)**

**Creational design Pattern: It simply helps to create objects in a consistent and controlled way, so making it easier to modify the creation process without affecting the rest of the system.**

**An example of a creational design pattern is Singleton Pattern.**

**Singleton Pattern basically restricts the instantiation of a class to single instance and provides a global point of access to that instance. So in a way it prevent redundancy and ensures that only one instance of a particular class can exist within an application.**

**An example of using it will be using the pattern in logging systems, in this system we use the singleton to capture and store information in one single location like using the pattern to ensure that all logging information is stored in one and single place rather than duplicating the data across multiple instances.**

**Structural design Pattern: This design pattern focuses on the composition of classes and objects to form larger structures, such as complex objects or whole subsystems.**

**We use this pattern to better organize our code to be more flexible, reusable, and maintainable.**

**An example of a structural design pattern is the Adapter Pattern.**

**The Adapter Pattern is used when we have two incompatible interfaces that need to work together, so it basically acts as a bridge between these two interfaces and helps them communicate with each other.**

**A real life example: depending on what I understood we use this pattern when we have for example two pieces of hardware(or software of course) that we want to connect to each other like connecting an old laptop with VGA port to a modern screen, both of them acts as interfaces and we want to connect them together and in this case we will have to get an adapter to interpret the signal from the VGA port to the HDMI port for example, so in this case the adapter it self represent the Adapter Pattern connecting the two interfaces together.**

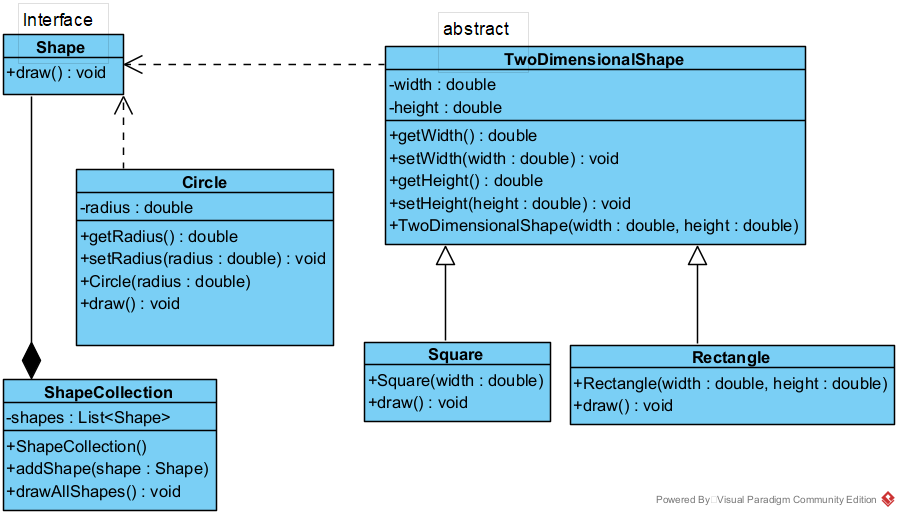
**Behavioural design Pattern: In general this pattern focuses on the interactions and communication between objects or classes.**

**An example of this pattern is the Command pattern.**

**Command pattern: In way the command pattern receive and store commands to be performed with its corresponding action or behaviour, which can be executed later on.**

**A real life example using this pattern would be A smart house system with voice activation commands, when the owner gives a command to for example turn on the lights, this command get translated into a command object that contains the necessary information, such as which light to be turned on, then that object is executed by the system which triggers the corresponding actions.**

**Task.5)**

****

**Explanation:**

**First we had the Shape interface which the “interface” Keyword in the code identified it as an interface, so how we know it was an interface, then the Shape interface had a method called draw and its return type was void so drew them in our UML and finished the interface.**

**After that we got the TwoDImensionalShape class which is abstract(we knew that by the abstract keyword before it) so we noted it as abstract then I noticed that it says that this class implements the Shape interface so I figured that there are a connection between them and that the TwoDimensionalShape class implements that interface,meaning it has to implement every method or variable in the interface in it’s own body, we represent this relationship in UML by a doted line, inside the class we have two Private attributes which are width and height both of them of type double so we wrote that and specified that both of them are private by drawing the “-” operation to indicate that they are private, then we wrote the setters and getters for those two attributes and then the parameterized constructor.**

**After that we have the Rectangle class which extends the TwoDimensionalShape class which means that it inherits all of the TwoDimensionalShape class behaviours and attributes, we represent this relationship as a line with an arrow shaped in the end pointing toward the parent class (in this case the TwoDimensionalShape class), in the Rectangle class we have the parametrized constructor and the method that exist in the interface called draw, we implemented the draw method from the interface here because we didn’t implemented it in the parent class so we implemented it here, so we get one method and a constructor in this class with the relationship of inheriting from another class.**

**After that we have the Square class which extends the TwoDimensionalShape class which means as I said earlier that this class have a relationship and acts as a subclass of the TwoDimensionalShape class, so we represent this relationship with a straight line with an arrow shaped at the end pointing at the super class ( in our case TwoDimensionalShape class), in the Square class we have a parametrized constructor, and the draw method from the interface that we must implement as we stated earlier, the method is public and have a void return type.**

**Next we have the Circle class which implements the Shape interface and as I say before the implement keyword in the code indicate that they have this relationship between an interface and a class, so we represent this relationship by a doted line with an arrow pointed to the interface, the class has one attribute which is radius of type double and of course it is private so we indicate that by the “-“ sign in the UML, then we put the setters and getters methods and the constructor, here we have one constructor which is parametrized one, with the class attribute(radius), after that we have the draw method which we had to implement due to the relationship between this class and the interface.**

**Lastly we have the ShapeCollection class which has one private attribute named shapes of type List<Shape>, then we have a default constructor with no parameters and two public methods named addShape with one parameter that takes in a variable called shape of type Shape, then the second method named drawAllShapes with no parameters and a public modifier.**

**After last 😊 we have the relationship between the interface and the SahpeCollection class as composition, we figured this out due to the class having a List of type Shape which is the interface so we conclude that it has a composition relationship cause the class has a list of Shape.**

**Plagiarism**

Plagiarism is a particular form of cheating. Plagiarism must be avoided at all costs and students who break the rules, however innocently, may be penalised. It is your responsibility to ensure that you understand correct referencing practices. As a university level student, you are expected to use appropriate references throughout and keep carefully detailed notes of all your sources of materials for material you have used in your work, including any material downloaded from the Internet. Please consult the relevant unit lecturer or your course tutor if you need any further advice.

**Student Declaration**

|  |
| --- |
| **Student declaration**  I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.  Student signature: Abdelrahaman Saleh Date:20/04/2023 |