Programming:

Section 1:

Object Oriented programming (OOP) is an approach in programming which includes or is mainly based on the concept of classes and objects. We tend to rely on it to structure a program into simple, reusable pieces of code (known as classes) that are used to create instances of objects.
There are several examples of object-oriented programming languages such as: Java, C++, and Python.
A programmer designs a program by collecting related pieces of data and behaviors together into one class. Then individual objects are instantiated from that class. The entire program consists of multiple objects interacting with objects to create the larger program.

OOP concepts:

Data Abstraction and Encapsulation:

Collecting related data and functions into one class is known as encapsulation. The data cannot be accessed from the outside world, and only those functions which are wrapped within the class are able to access it. These functions act as the interface between the object's details and the program itself. This insulation of the data (program can not access it directly) is known as data hiding or information hiding.

Advantages of Encapsulation:

•**Data Hiding:** The user wont have any background about the details of the inner implementation of the class. It is not to be visible to the user how the class stores values in the variables. The user only knows that we are passing the values using setter methods and variables are then initialized with that value.

•**More Flexibility:** We have the choice to make the variables of the class read-only or write-only depending on what we require. If we wish to make the variables as read-only then we remove setters or if we wish to make the variables as write-only then we remove getters.

•**Reusability:** Encapsulation also improves the re-usability and makes it to change as new requirements occur.

•**Easy to test code:** easy to test encapsulated code for unit testing.

Abstraction is a term that describes the paradigm of representing essential features without including the background details or any explanations. Classes use this concept and are defined as a list of abstract attributes such as size, weight, and cost, and the methods operate on these attributes. They encapsulate all the needed details of the object that is to be created.

Advantages of Abstraction

- It reduces the complexity.
- Code duplication is avoided and enhances reusability.
- Makes an application or program more secure as only important details are provided to the user.

When should we use Abstract Methods & Abstract Class?
Abstract methods are mostly used when two or more subclasses are doing the exact thing however in different ways through different implementations. It also extends the same Abstract class and gives a variety of implementations of the abstract methods.
Abstract classes is a good way to describe behaviors that are generic and object-oriented programming class hierarchy. Not only this it also describes subclasses offering implementation details of the abstract class.

Encapsulation vs Data Abstraction

- Encapsulation can be described as data hiding(information hiding) while Abstraction is about detail hiding(implementation hiding).
- Encapsulation collects together data and methods that act upon the data, data abstraction is about exposing the interface but hiding the implementation details.

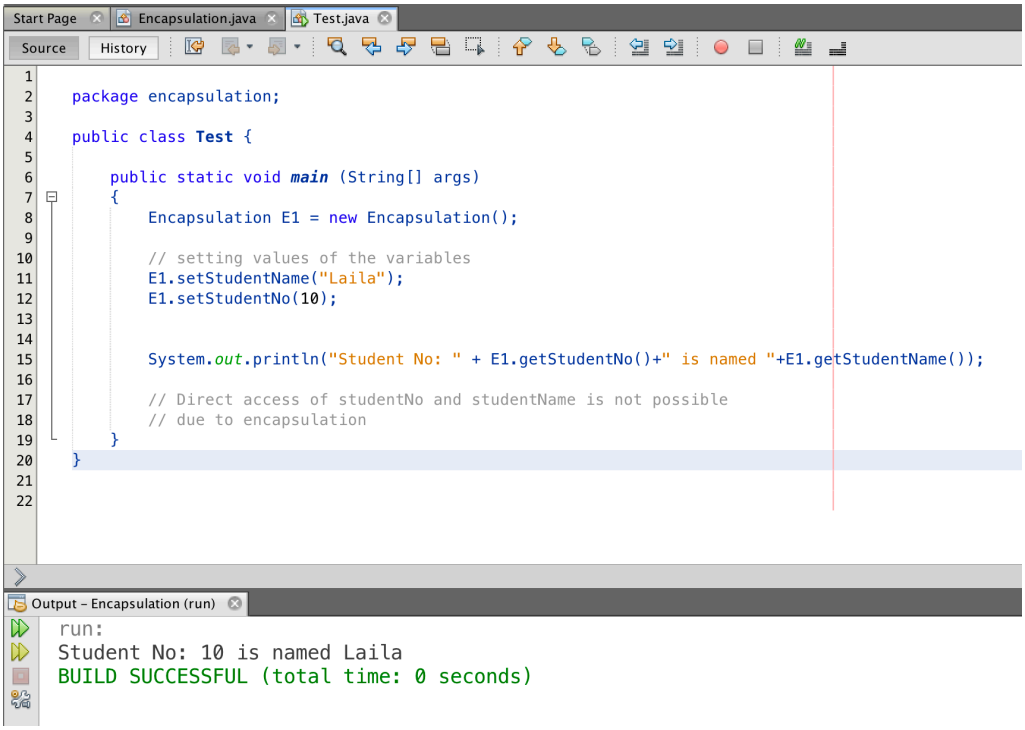
An easy example to comprehend this difference is a mobile phone. Where the logic in the circuit board that is complex is encapsulated in a touch screen, and the interface is provided to perform the role of abstracting it out.

Realization of Encapsulation in Java :

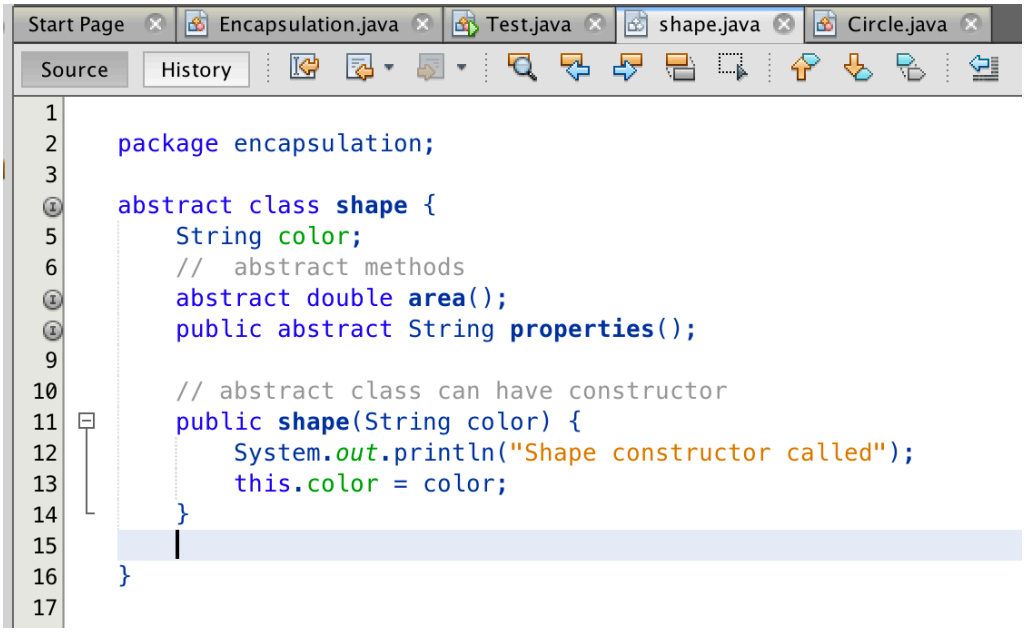
Start Page | Encapsulation.java | Test.java

Source | History

```
1
2   package encapsulation;
3
4       // Java program to demonstrate encapsulation
5   public class Encapsulation
6   {
7      //private variables declared that can only be accessed by public class methods
8       private String studentName;
9       private int studentNo;
10
11      public String getStudentName()
12      {
13        return studentName;
14      }
15
16      public int getStudentNo()
17      {
18         return studentNo;
19      }
20
21      public void setStudentName(String Name)
22      {
23        studentName = Name;
24      }
25
26      public void setStudentNo( int No)
27      {
28        studentNo = No;
29      }
30  }
```



Realization of Abstraction in Java :



Start Page | Encapsulation.java | Test.java | shape.java | Circle.java

Source | History

```java
package encapsulation;

class Circle extends shape
{
    double radius;

    public Circle(String color,double radius) {

        // calling Shape constructor
        super(color);
        System.out.println("Circle constructor called");
        this.radius = radius;
    }

    @Override
    double area() {
        return 3.14 * radius * radius;
    }

    @Override
    public String properties() {
        return "Circle color is " + super.color +
                    " and area is : " + area();
    }


}
```

Start Page | Encapsulation.java | Test.java | shape.java | Circle.java

Source | History

```java
package encapsulation;

public class Test {

    public static void main (String[] args)
    {
        //Encapsulation E1 = new Encapsulation();
        // setting values of the variables
       // E1.setStudentName("Laila");
       // E1.setStudentNo(10);
        //System.out.println("Student No: " + E1.getStudentNo
        // Direct access of studentNo and studentName is not
        // due to encapsulation
        shape s1 = new Circle("Red", 2);
        System.out.println(s1.properties());

    }
}
```

Output – Encapsulation (run)

```
run:
Shape constructor called
Circle constructor called
Circle color is Red and area is : 12.56
BUILD SUCCESSFUL (total time: 0 seconds)
```

Inheritance:

Inheritance **is the method** that involves objects **of 1 class obtaining** the properties of objects of another **classes**. It acts as a support for the concept of hierarchical classification. The principal behind **this kind** of division is **that every** derived **class** shares common features with **the class** from **that it's** derived from.
In OOP, inheritance provides **the thought** of reusability. **this suggests** that **we will** add **extra options** existing **classes while not** modifying it. **this is often attainable** by **getting a new class** from **the** one that exists. The new **class can** have the combined feature of **each** the **classes**. **the important charm** and power of the inheritance mechanism is that it **permits** the **programmer** to **use a class again** i.e almost, **however** not **specifically**, what he **needs**, and to tailor **the class** in such **the way** that it **doesn't** introduce any undesirable side-effects into **the remainder** of **the classes**.

Advantages of Inheritance

- We are able to use code written more than once, i.e. code reusability.
- One superclass can be used for all the subclasses in a hierarchy.
- No changes are needed to be done in base classes, we just do changes in the parent class.
- Inheritance **is employed to get additionally** dominant objects.
- Inheritance is a good way to avoid duplicity and redundancy of data.
- Inheritance is a good way to avoid space complexity as well as time complexity.

When Should we Use Inheritance?
Mainly we use inheritance when we want to use specific parts of code and modify specific features according to our need and this is to be done without any complexity. Also when we need flexibility in our code to reuse it from base class to required derived class. A child class is able to override properties of base class without the need to rewrite code in the same class over and over.

Realization of Inheritance in Java :

Start Page | Encapsulation.java | Test.java | shape.java | Circle.java | Animal.java

Source | History

```
1   class Animal {
2
3       public void eat() {
4           System.out.println("I eat");
5       }
6
7   }
8
9   class Dog extends Animal {
10      public void sound() {
11          System.out.println("I bark");
12      }
13  }
14
15  class Main {
16      public static void main(String[] args) {
17          Dog d1 = new Dog();
18          d1.eat();
19          d1.sound();
20      }
21  }
```

Output - final (run)

```
run:
I eat
I bark
BUILD SUCCESSFUL (total time: 0 seconds)
```

Polymorphism:

Polymorphism is another important concept of object oriented programming. Polymorphism, a Greek word, that means the ability to take several forms. An operation may adopt different behavior in different instances. The behavior is dependent upon the types of data that are used in the operation. For example, lets analyze the addition operation. When we have two numbers, the operation will result in a sum. However if the operands are strings, then the operation produces a third string using concatenation. The process of making an operator adopt different behaviors in different instances is called operator overloading.

This is something close to a word having several different meanings in different contexts. Using the same function name to do different types of tasks is called **function overloading.**

Polymorphism plays a vital role in allowing objects with different internal structures to have the same external interface. This means that a general class of operations can be accessed in the same way even if specific actions related to each operation are different. Polymorphism is widely used in implementing the concept of inheritance.

Advantages

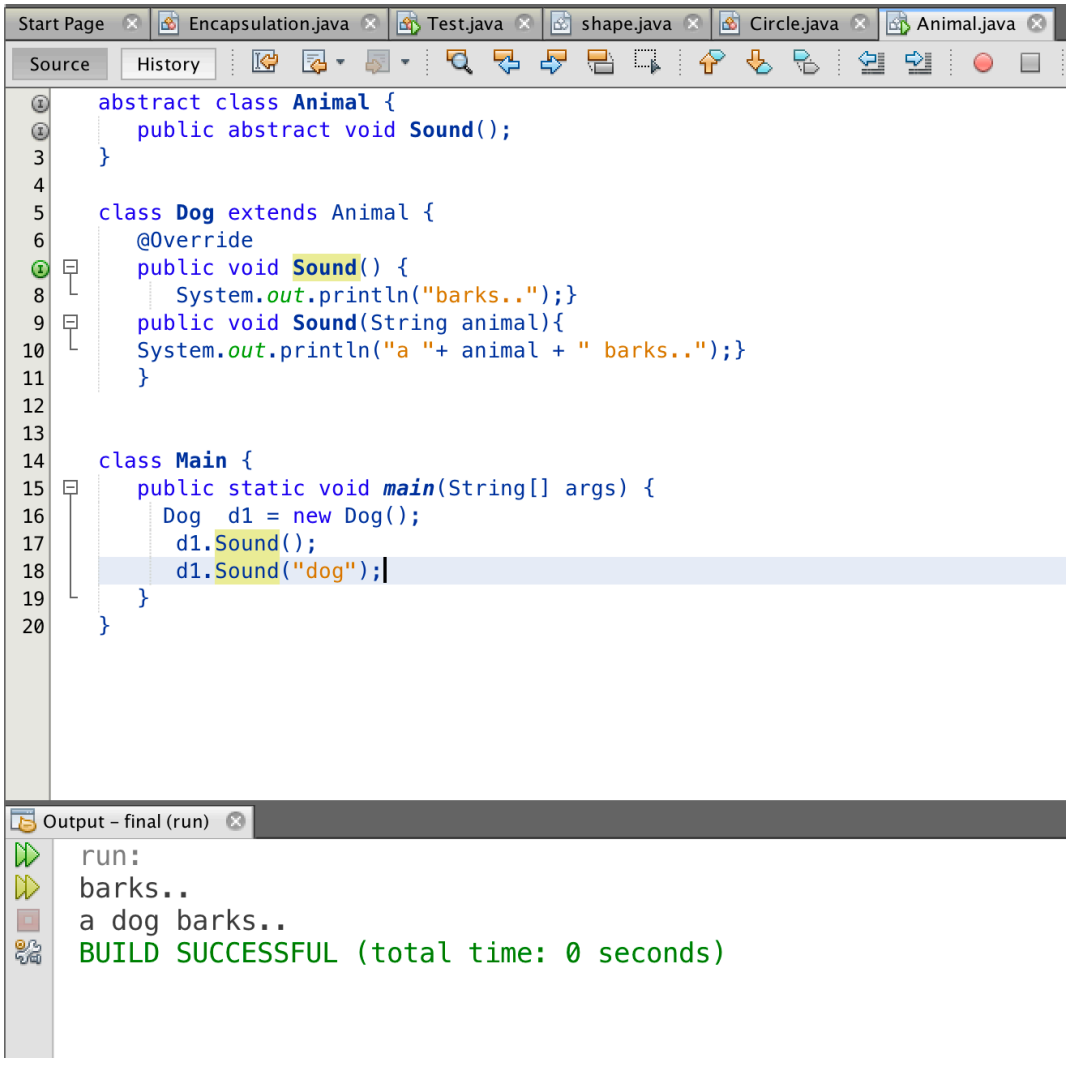
•Reusability: Polymorphism enables programmers to reuse the code and also the classes that are once written.

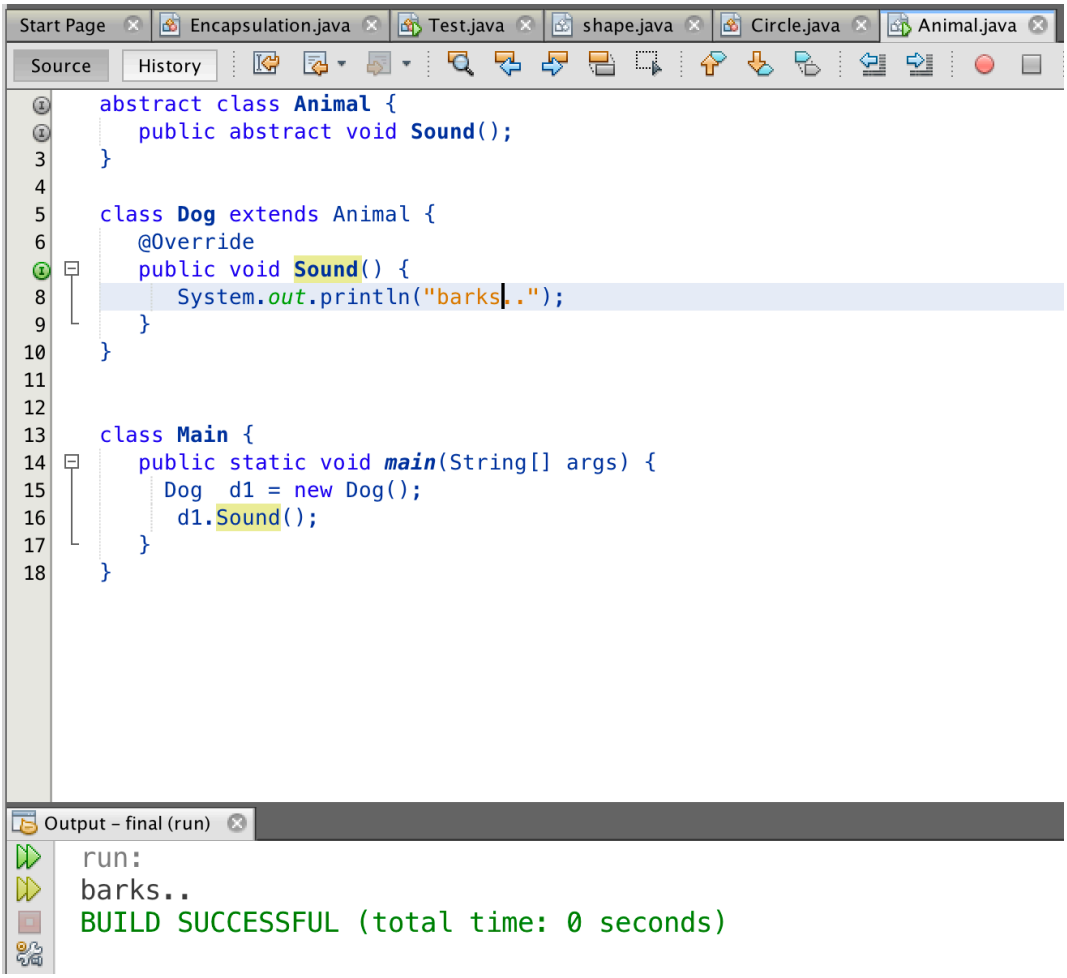
•One variable name can hold variables of multiple data types.

•Enhances program readability.

When do we need polymorphism?
Polymorphism is needed as this concept is widely used in implementing inheritance. We need when we want objects with varied internal structures to share the same external interface. [1]

Realization of Polymorphism in Java :

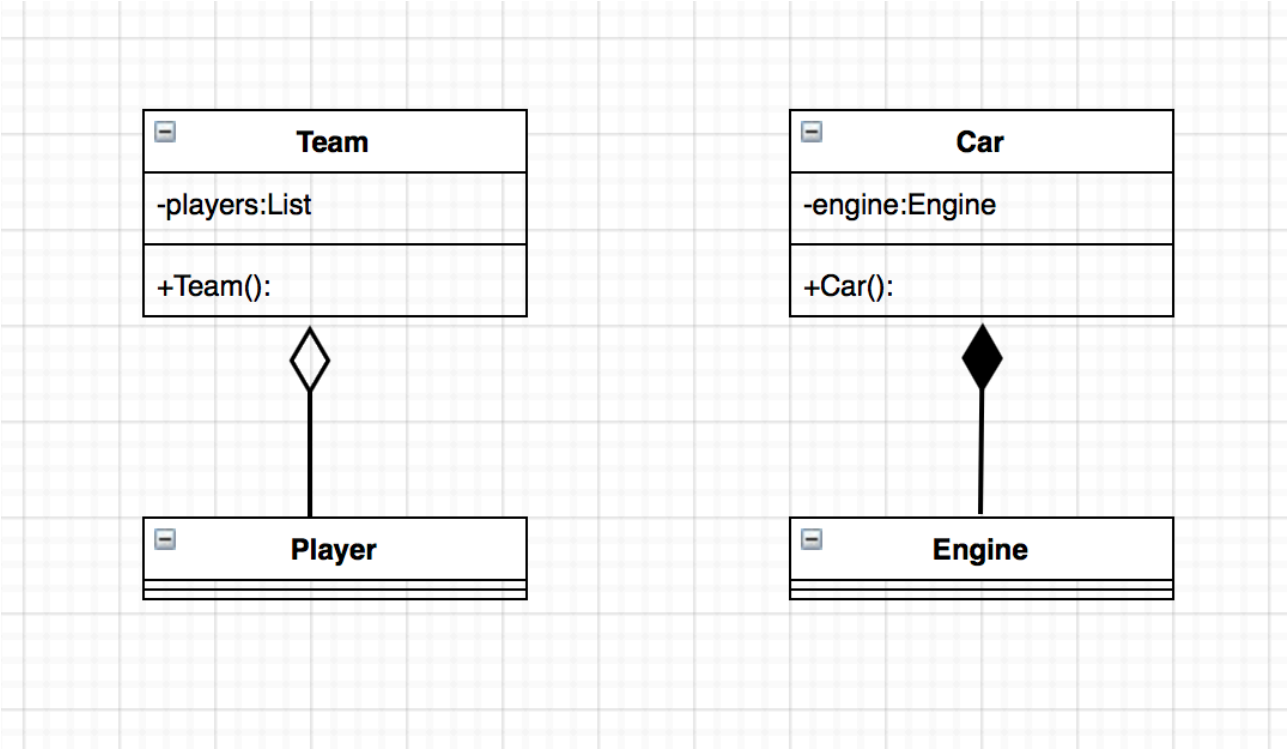



Start Page
Encapsulation.java
Test.java
shape.java
Circle.java
Animal.java
Source
History
abstract class Animal {
public abstract void Sound();
}
class Dog extends Animal {
@Override
public void Sound() {
System.out.println("barks..");
}
}
class Main {
public static void main(String[] args) {
Dog d1 = new Dog();
d1.Sound();
}
}
Output – final (run)
run:
barks..
BUILD SUCCESSFUL (total time: 0 seconds)

Section 2:

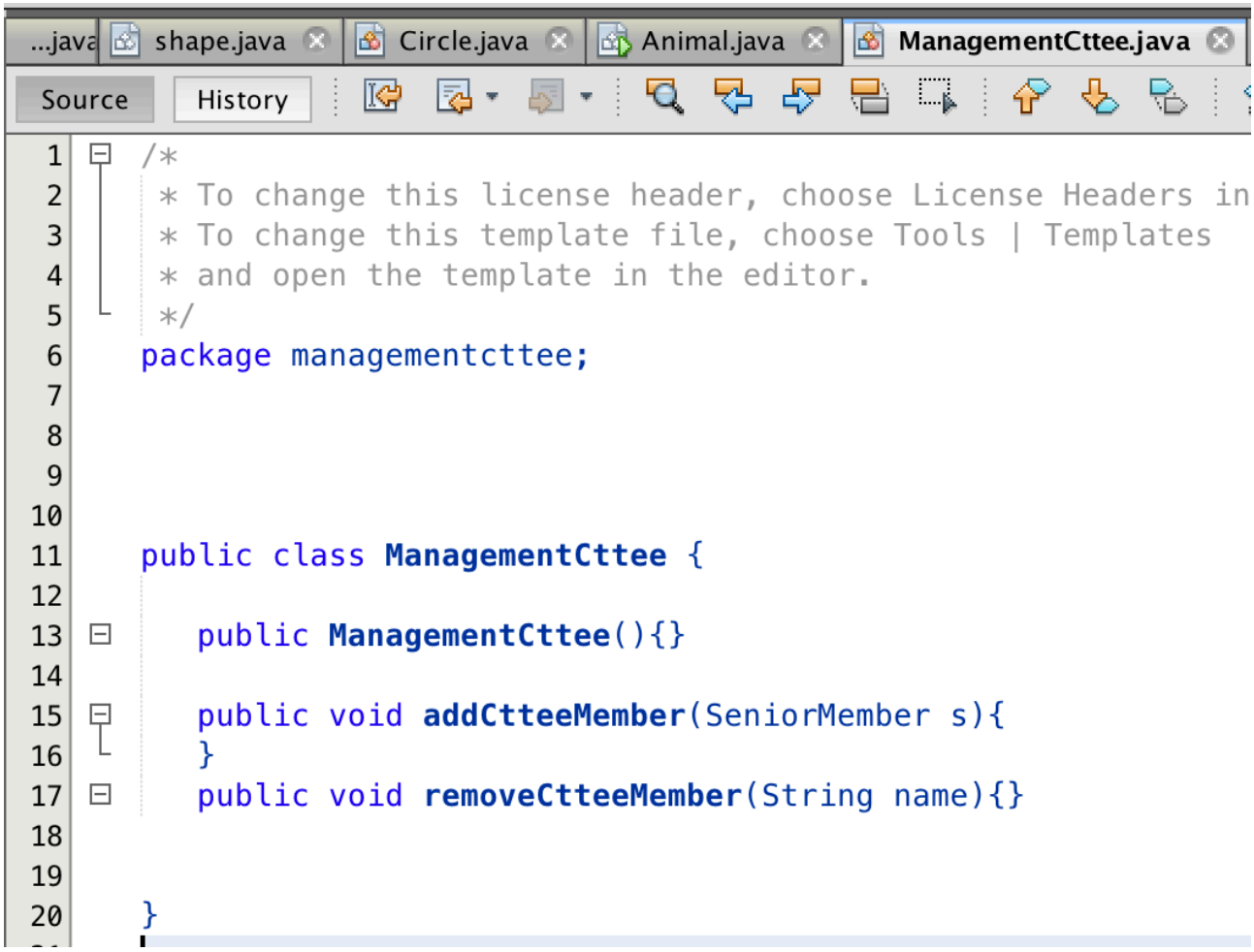
a-

I. Composition relationship
II. Aggregation relationship
III.



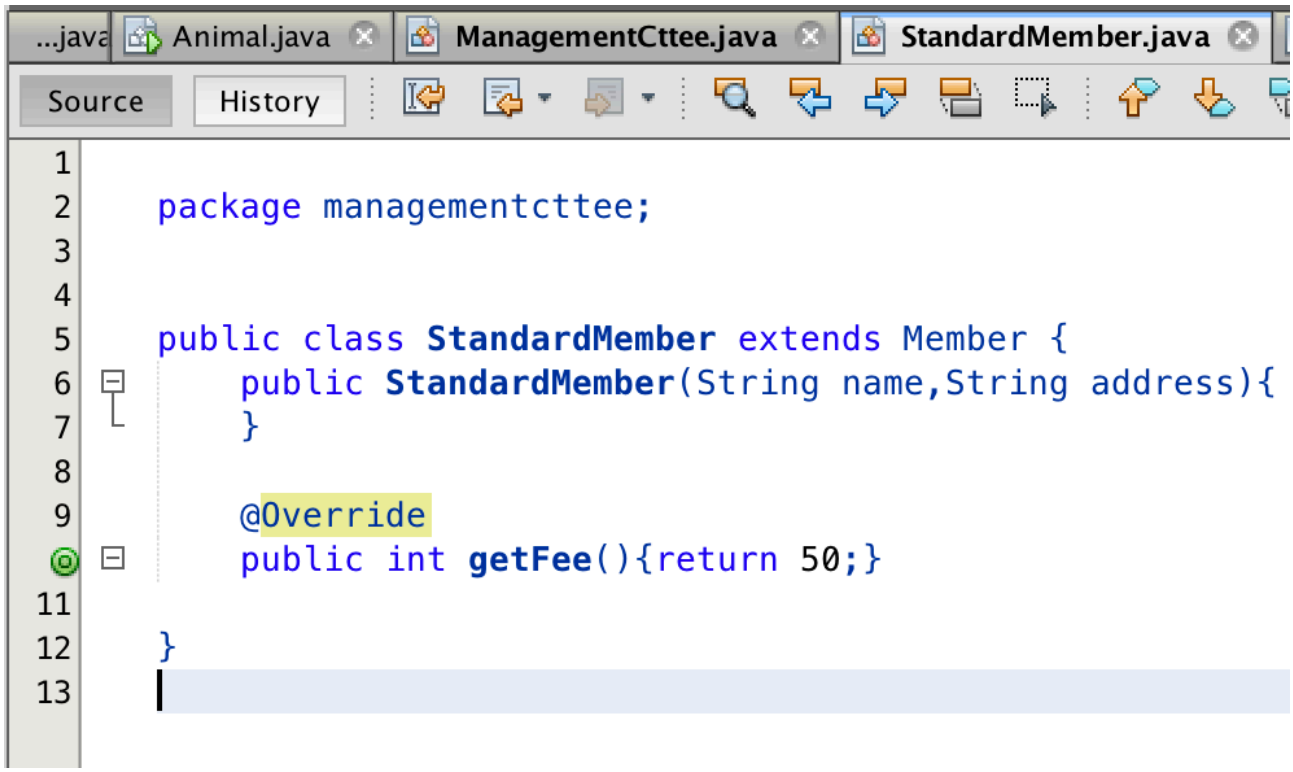
b-

I.

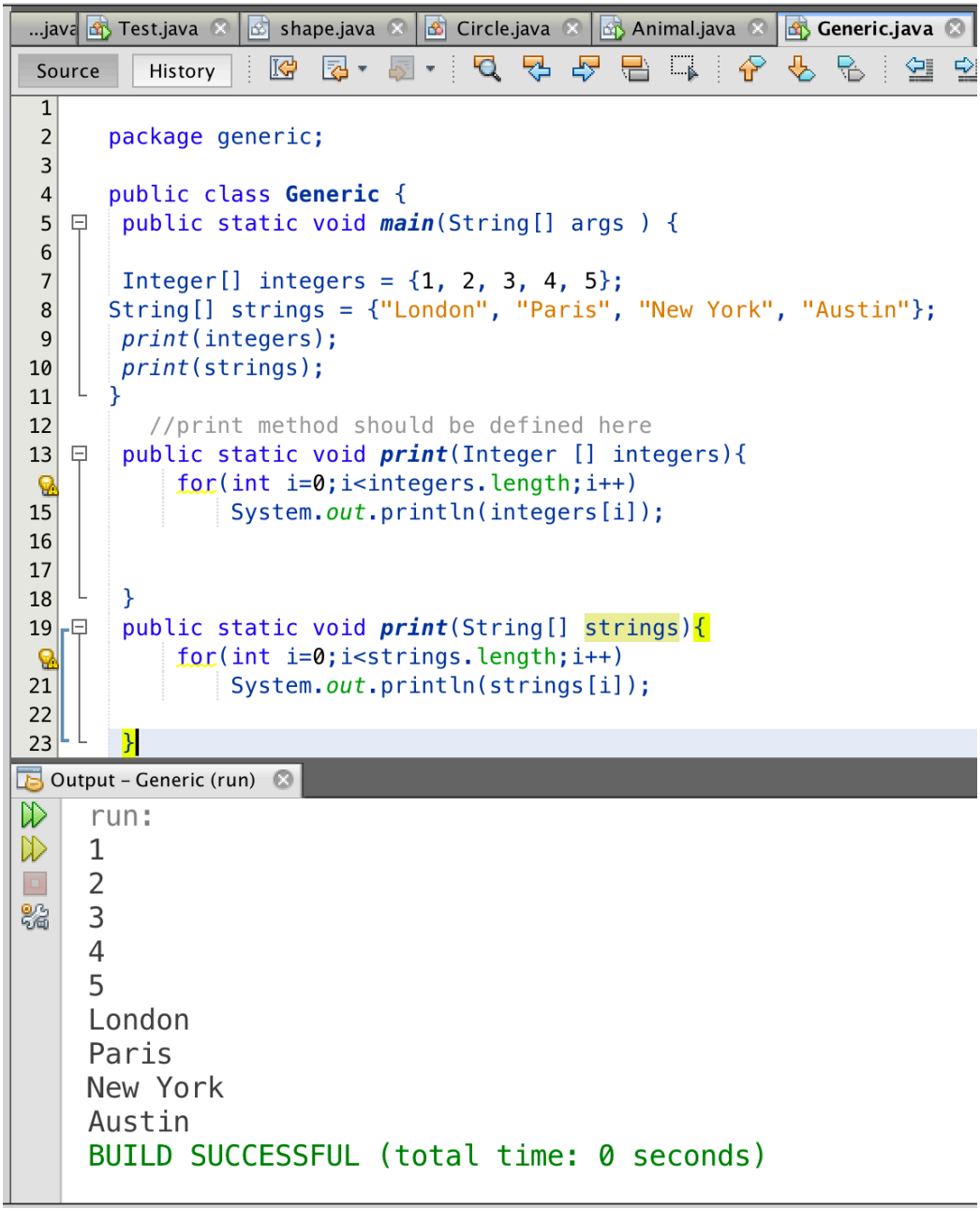


II. An abstract class is a type of class that is special as we cannot instantiate it. It is made so that it is inherited by subclasses that either implement its methods or override them. To make it simple we can say, abstract classes are either partially implemented or are not implemented at all. An abstract class may contain constructors that is one of the main differences between abstract classes and interfaces. As for the interface it is not a class, However it is very close to an abstract class. It contains methods without a body. It doesn't have the ability to do anything on its own. Like the abstract class it is used to impose guidelines and hierarchies and contain methods to be used by the sub classes. One instant in which we find interfaces to be really handy is when: A class is not able to inherit from several abstract classes at one time in languages such as Java and C. Because of their lack of support for the concept of multiple-inheritance, that is when we get to use interfaces. An interface is known as the declaration of methods of an object only; it is not the implementation itself. In an interface, we define then kind of operation the class is able to perform. These operations are then defined by the classes implementing the interface.

III.

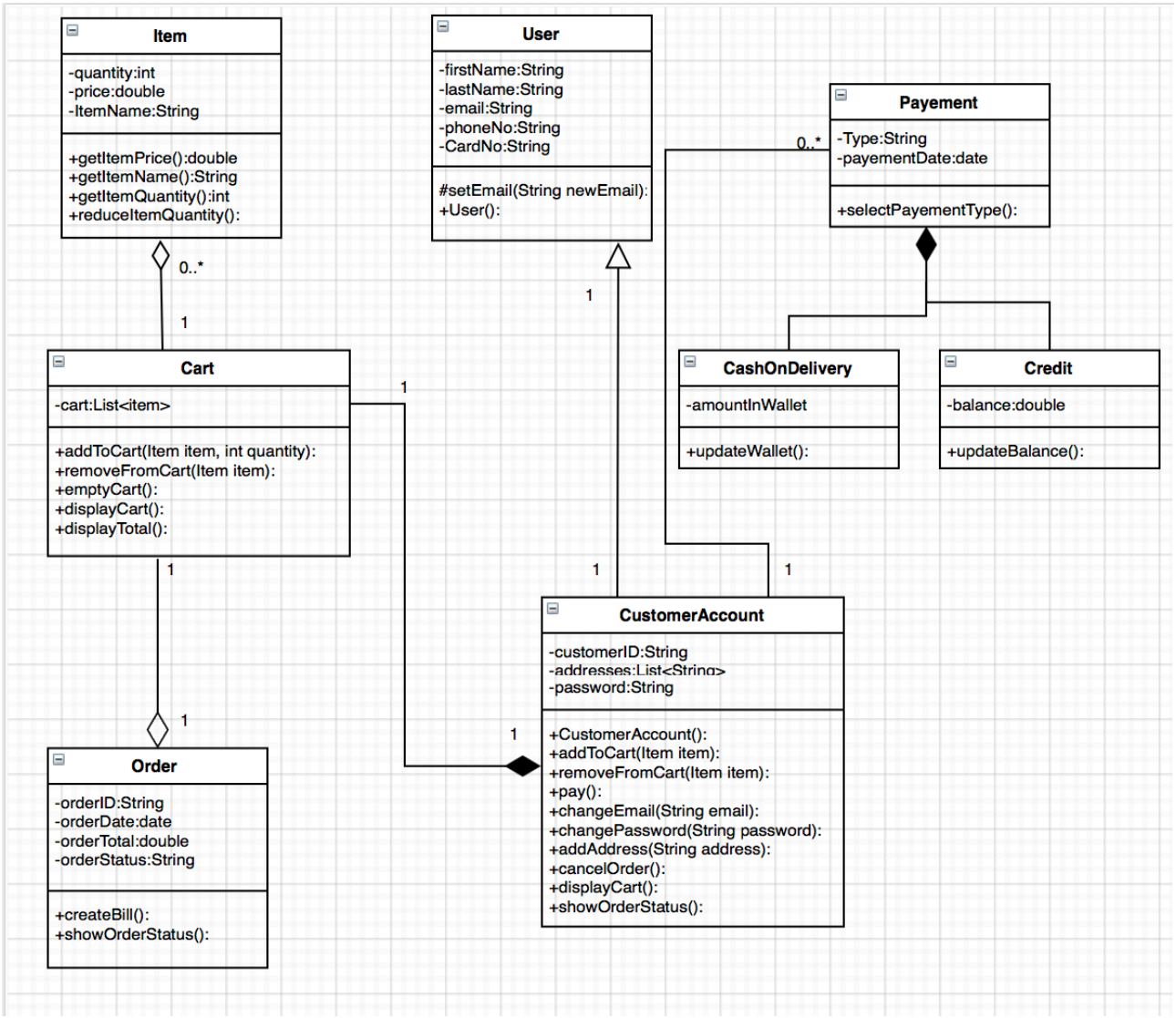

...java
Animal.java
ManagementCttee.java
StandardMember.java
Source
History
1
2 package managementcttee;
3
4
5 public class StandardMember extends Member {
6 public StandardMember(String name,String address){
7 }
8
9 @Override
public int getFee(){return 50;}
11
12 }
13

c-


...java
Test.java
shape.java
Circle.java
Animal.java
Generic.java
Source
History
1 package generic;
2
3 public class Generic {
4 public static void main(String[] args) {
5
6 Integer[] integers = {1, 2, 3, 4, 5};
7 String[] strings = {"London", "Paris", "New York", "Austin"};
8 print(integers);
9 print(strings);
10 }
11 //print method should be defined here
12 public static void print(Integer [] integers){
13 for(int i=0;i<integers.length;i++)
14 System.out.println(integers[i]);
15
16
17 }
18 public static void print(String[] strings){
19 for(int i=0;i<strings.length;i++)
20 System.out.println(strings[i]);
21
22 }
23
Output - Generic (run)
run:
1
2
3
4
5
London
Paris
New York
Austin
BUILD SUCCESSFUL (total time: 0 seconds)

Section 4-

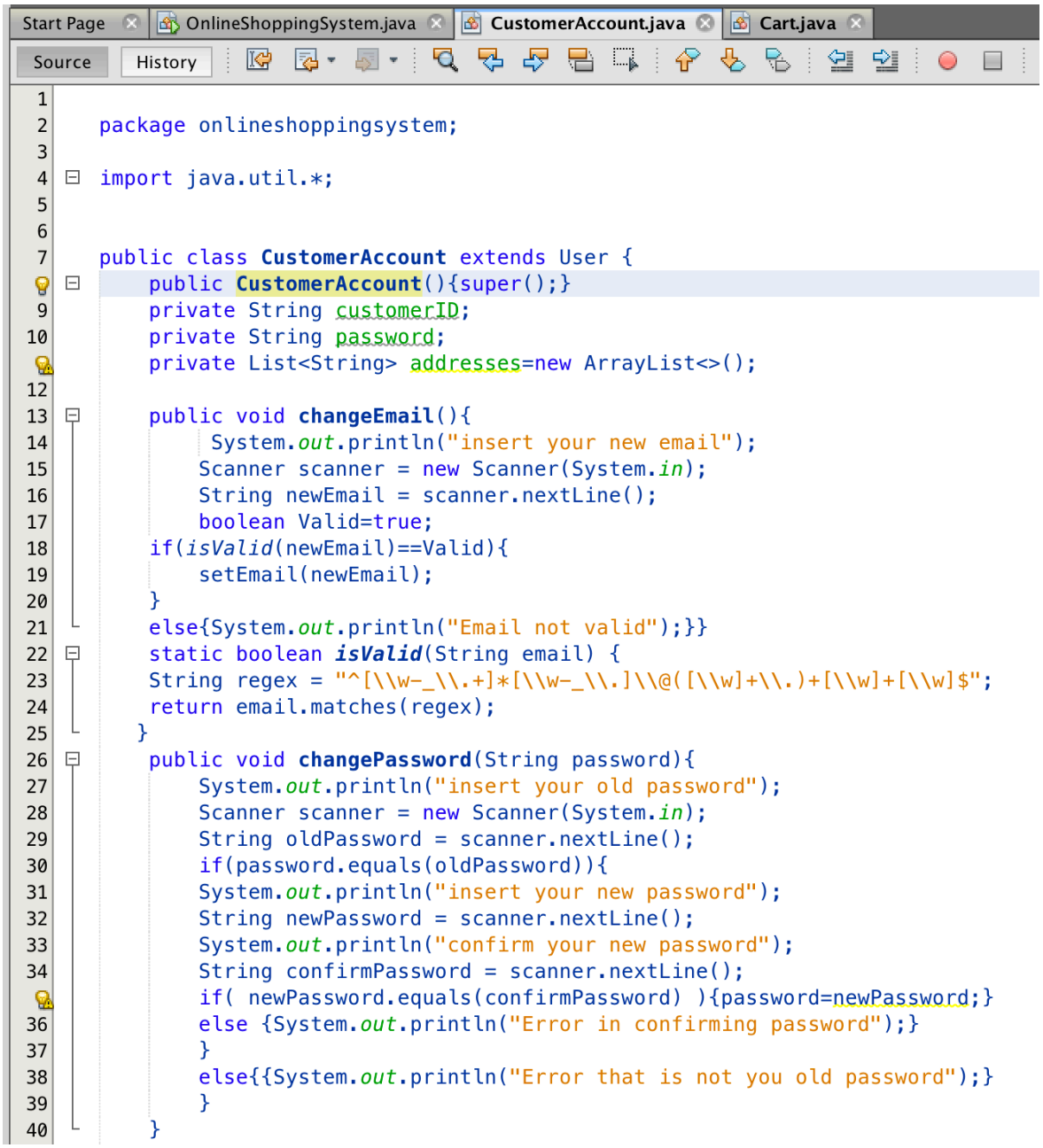
A-



B- the two main concepts used in my design are Inheritance and Encapsulation.
I used Inheritance as class CustomerAccount inherits class User. Class Customer Account shares common properties with class User so instead of rewriting these properties one class inherits the other that way we avoid redundancy of data as well as being able to reuse code later.In my code it can be seen that the CustomerAccount constructor calls the User constructor using the word super. Also dividing classes that way reduces complexity.
I used Encapsulation which means that I collected all the related data and functions in the same class. Also I made all the attributes private which means they cant be accessed from outside the class. And there are functions that are responsible for changing and updating these attributes. This concept enhances code reusability and improves security as the attributes are hidden so the implementation and logic are more secure.

C-Two of the main classes of my design are the CustomerAccount class and the Cart class

CustomerAccount class complete code:

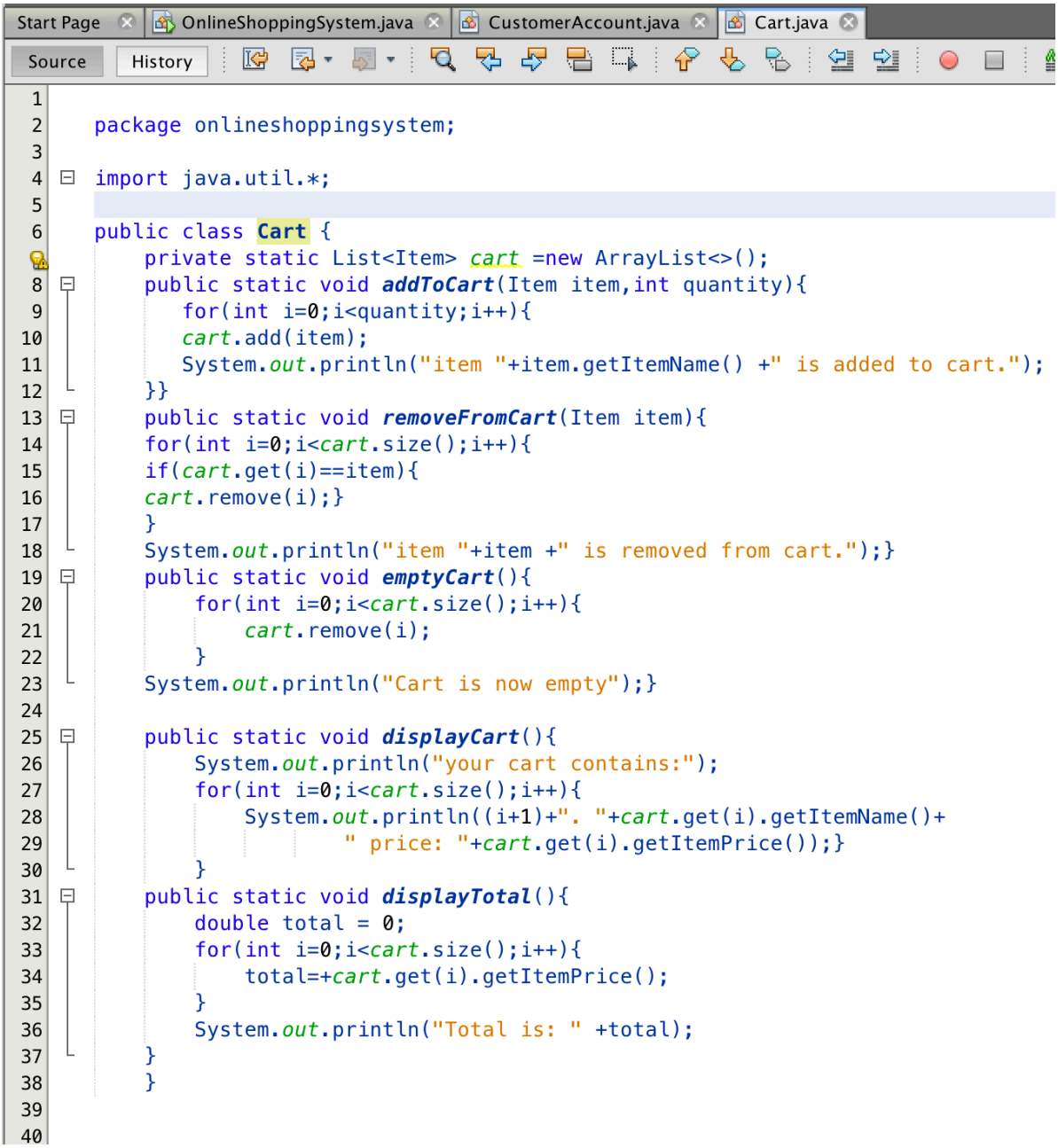


Start Page | OnlineShoppingSystem.java | CustomerAccount.java | Cart.java

Source | History

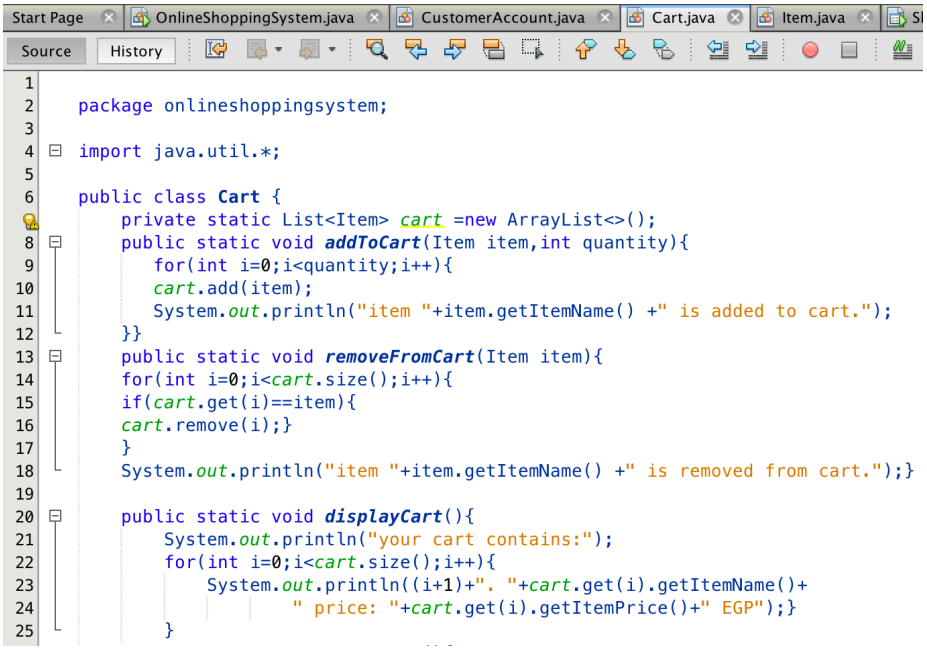
```
40      }
41      public void addAddress(){
42      System.out.println("insert the Building Number");
43      Scanner scanner = new Scanner(System.in);
44      String buildingNo = scanner.nextLine();
45      System.out.println("insert the Street Name");
46      String streetName = scanner.nextLine();
47      System.out.println("insert the Neighborhood");
48      String neighborhood = scanner.nextLine();
49      System.out.println("insert the City");
50      String city = scanner.nextLine();
51      System.out.println("insert the Postal Code");
52      String postalCode = scanner.nextLine();
53      String newAddress = buildingNo+", "+streetName+", "+neighborhood+", "
54              +city+", "+postalCode+".";
55      addresses.add(newAddress);}
56      public void addToCart(Item item,int quantity){
57      Cart.addToCart(item,quantity);}
58      public void removeFromCart(Item item){
59      Cart.removeFromCart(item);}
60      public void displayCart(){
61      Cart.displayCart();}
62      public void pay(){
63      boolean billCreated=Order.createBill();
64      boolean payementCompleted=Payement.selectPayementType();
65      if(billCreated==true && payementCompleted==true)
66      {System.out.println("Payement successful, you can now check your order status");
67      Cart.emptyCart();}}
68      public void showOrderStatus(){
69      Order.showOrderStatus();}
70      public void cancelOrder(){
71      Cart.emptyCart();}
72
73
74
75
76  }
77
```

Cart class complete code:

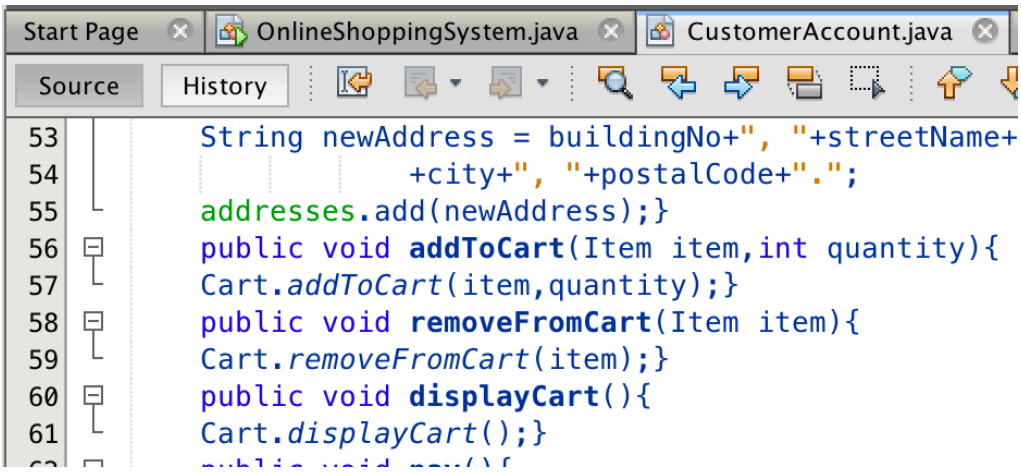



```
1
2   package onlineshoppingsystem;
3
4   import java.util.*;
5
6   public class Cart {
7       private static List<Item> cart =new ArrayList<>();
8       public static void addToCart(Item item,int quantity){
9          for(int i=0;i<quantity;i++){
10         cart.add(item);
11         System.out.println("item "+item.getItemName() +" is added to cart.");
12      }}
13      public static void removeFromCart(Item item){
14      for(int i=0;i<cart.size();i++){
15      if(cart.get(i)==item){
16      cart.remove(i);}
17      }
18      System.out.println("item "+item +" is removed from cart.");}
19      public static void emptyCart(){
20          for(int i=0;i<cart.size();i++){
21              cart.remove(i);
22          }
23      System.out.println("Cart is now empty");}
24
25      public static void displayCart(){
26          System.out.println("your cart contains:");
27          for(int i=0;i<cart.size();i++){
28              System.out.println((i+1)+". "+cart.get(i).getItemName()+
29                      " price: "+cart.get(i).getItemPrice());}
30          }
31      public static void displayTotal(){
32          double total = 0;
33          for(int i=0;i<cart.size();i++){
34              total=+cart.get(i).getItemPrice();
35          }
36          System.out.println("Total is: " +total);
37      }
38      }
39
40
```

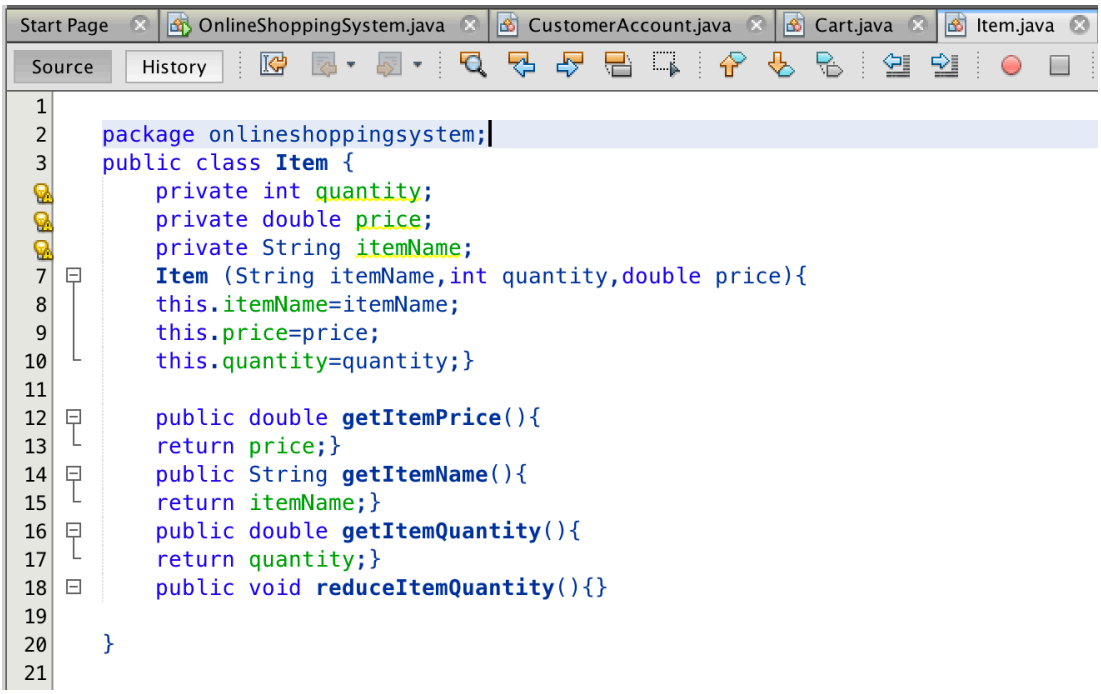
D-one of the main functionalities of the system is for the customer to add an item to cart and remove an item from cart.



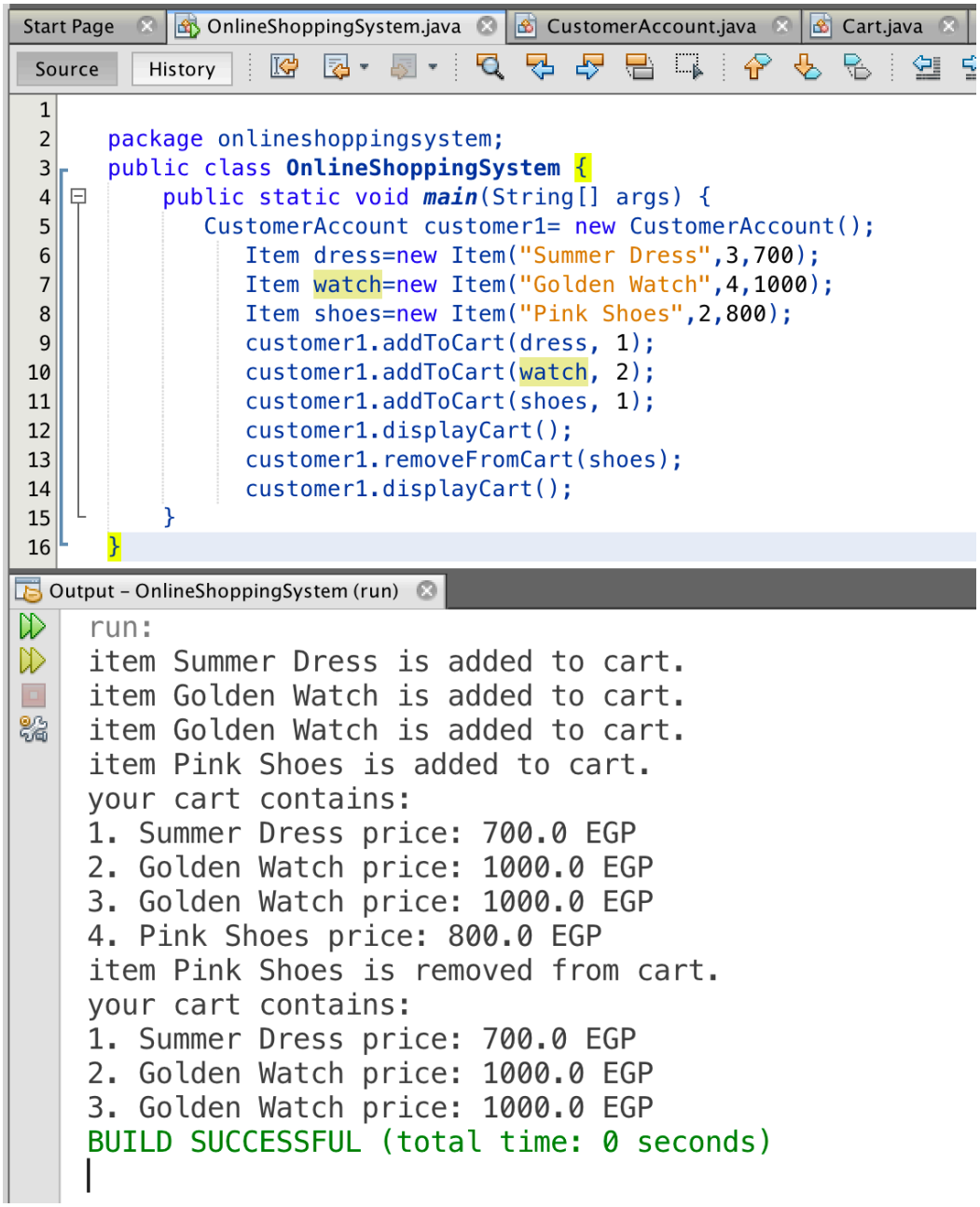
```java
package onlineshoppingsystem;

import java.util.*;

public class Cart {
    private static List<Item> cart =new ArrayList<>();
    public static void addToCart(Item item,int quantity){
       for(int i=0;i<quantity;i++){
       cart.add(item);
       System.out.println("item "+item.getItemName() +" is added to cart.");
    }}
    public static void removeFromCart(Item item){
    for(int i=0;i<cart.size();i++){
    if(cart.get(i)==item){
    cart.remove(i);}
    }
    System.out.println("item "+item.getItemName() +" is removed from cart.");}

    public static void displayCart(){
        System.out.println("your cart contains:");
        for(int i=0;i<cart.size();i++){
            System.out.println((i+1)+". "+cart.get(i).getItemName()+
                    " price: "+cart.get(i).getItemPrice()+" EGP");}
        }
```



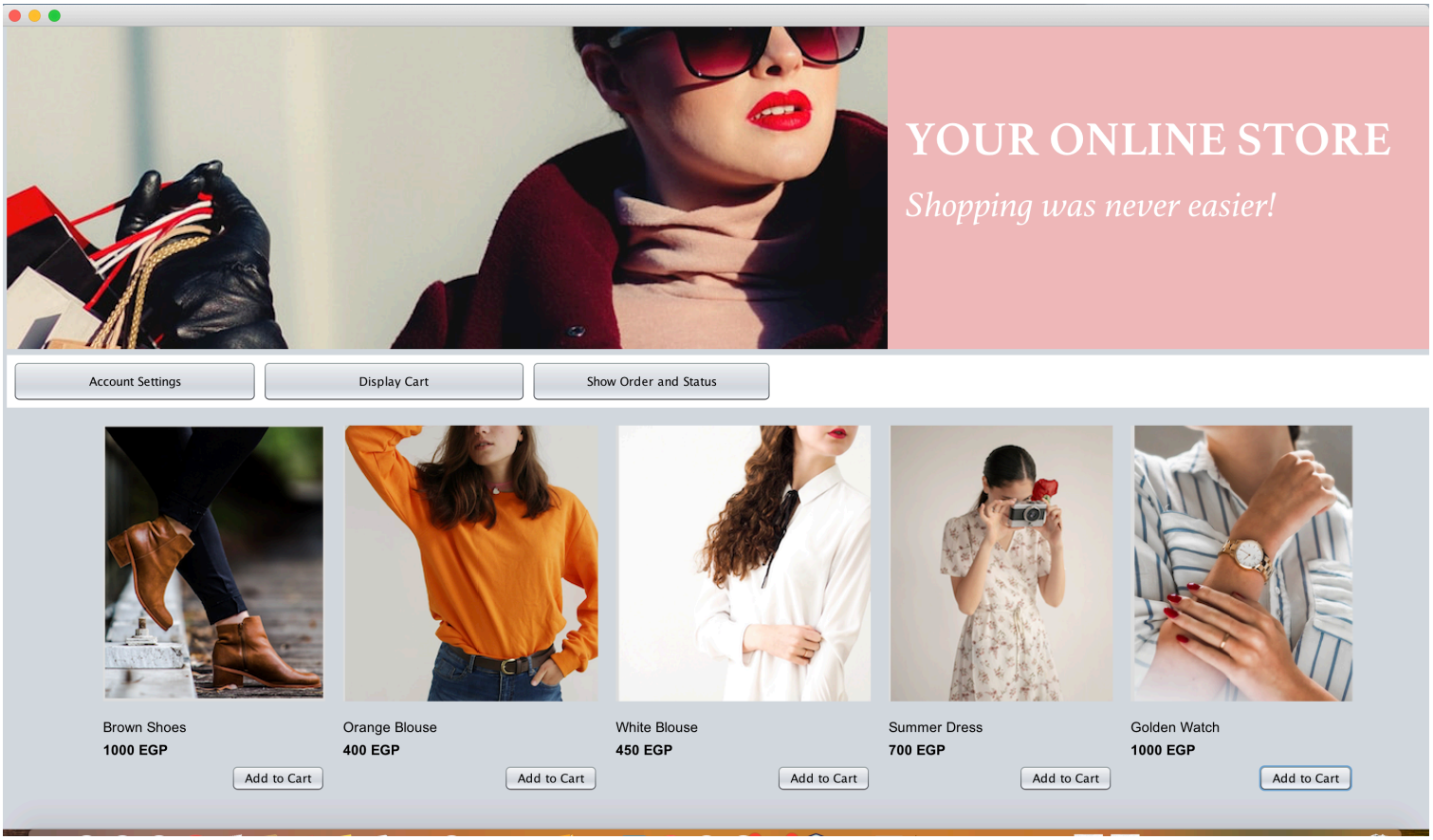
```java
        String newAddress = buildingNo+", "+streetName+
                +city+", "+postalCode+".";
        addresses.add(newAddress);}
    public void addToCart(Item item,int quantity){
    Cart.addToCart(item,quantity);}
    public void removeFromCart(Item item){
    Cart.removeFromCart(item);}
    public void displayCart(){
    Cart.displayCart();}
```



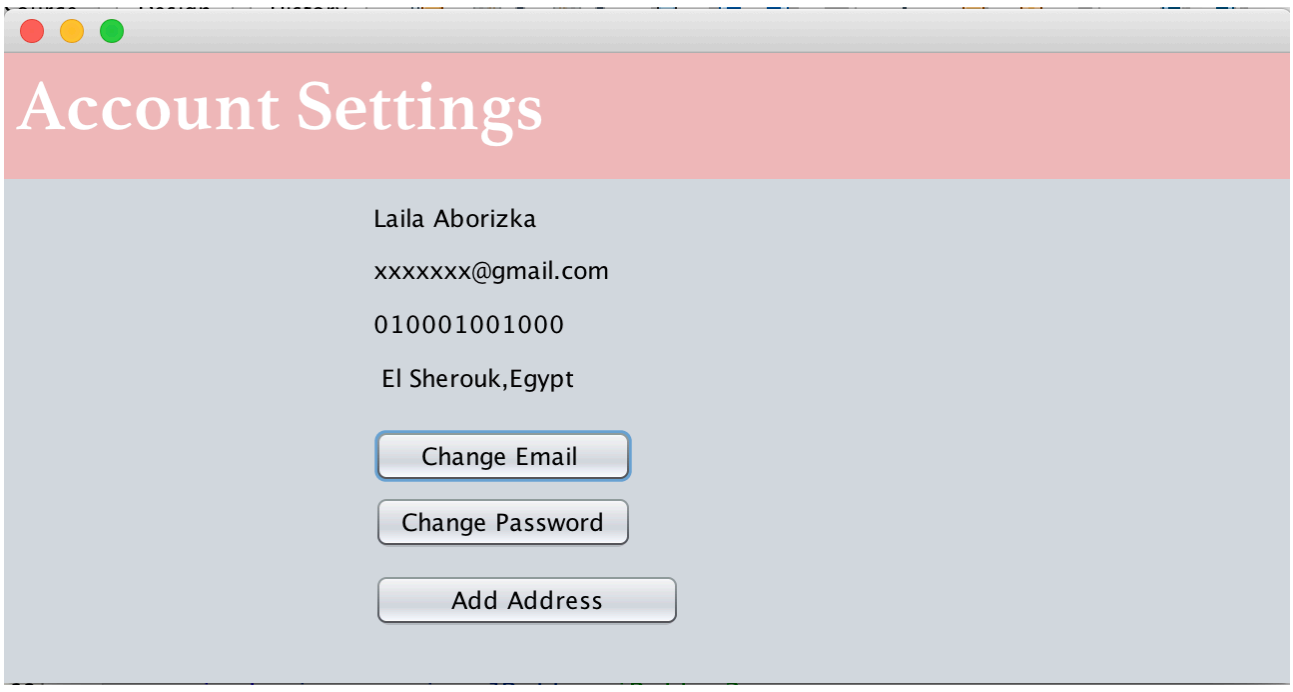
```java
package onlineshoppingsystem;
public class Item {
    private int quantity;
    private double price;
    private String itemName;
    Item (String itemName,int quantity,double price){
    this.itemName=itemName;
    this.price=price;
    this.quantity=quantity;}

    public double getItemPrice(){
    return price;}
    public String getItemName(){
    return itemName;}
    public double getItemQuantity(){
    return quantity;}
    public void reduceItemQuantity(){}

}
```


Start Page
OnlineShoppingSystem.java
CustomerAccount.java
Cart.java
Source
History
1
2 package onlineshoppingsystem;
3 public class OnlineShoppingSystem {
4 public static void main(String[] args) {
5 CustomerAccount customer1= new CustomerAccount();
6 Item dress=new Item("Summer Dress",3,700);
7 Item watch=new Item("Golden Watch",4,1000);
8 Item shoes=new Item("Pink Shoes",2,800);
9 customer1.addToCart(dress, 1);
10 customer1.addToCart(watch, 2);
11 customer1.addToCart(shoes, 1);
12 customer1.displayCart();
13 customer1.removeFromCart(shoes);
14 customer1.displayCart();
15 }
16 }
Output – OnlineShoppingSystem (run)
run:
item Summer Dress is added to cart.
item Golden Watch is added to cart.
item Golden Watch is added to cart.
item Pink Shoes is added to cart.
your cart contains:
1. Summer Dress price: 700.0 EGP
2. Golden Watch price: 1000.0 EGP
3. Golden Watch price: 1000.0 EGP
4. Pink Shoes price: 800.0 EGP
item Pink Shoes is removed from cart.
your cart contains:
1. Summer Dress price: 700.0 EGP
2. Golden Watch price: 1000.0 EGP
3. Golden Watch price: 1000.0 EGP
BUILD SUCCESSFUL (total time: 0 seconds)

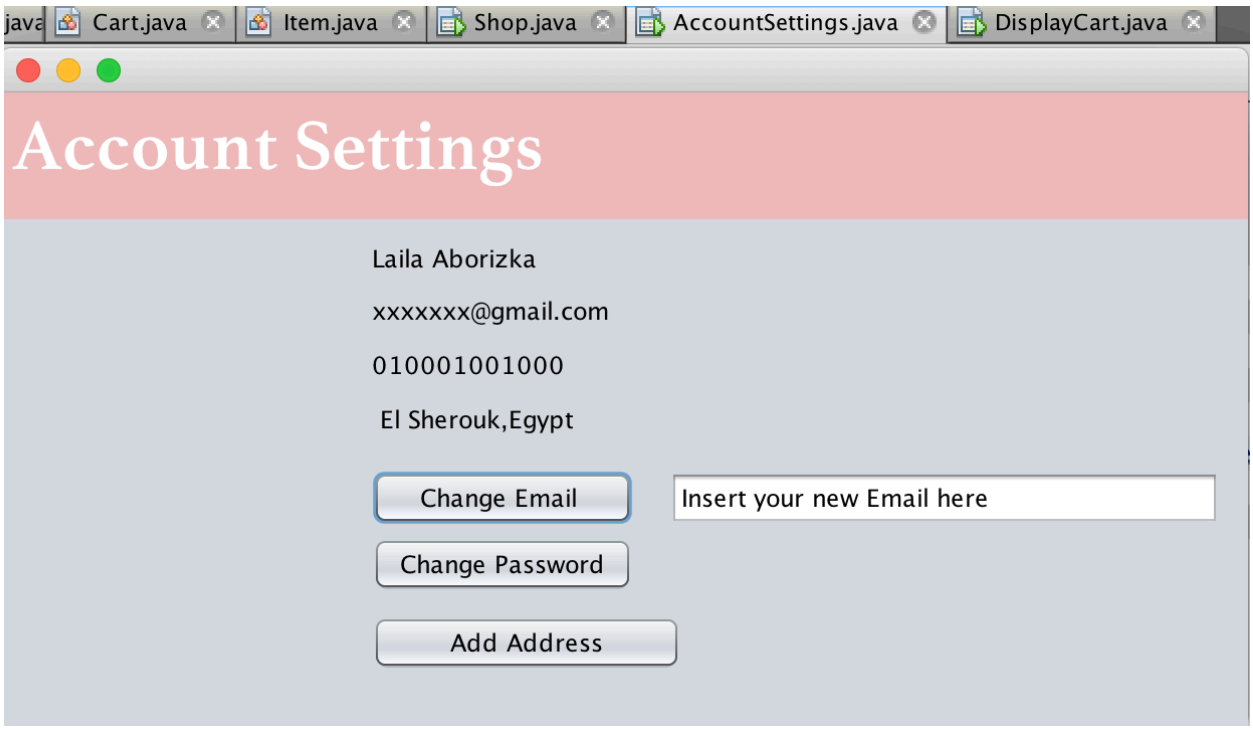
E-the UI screenshots:
The Homepage: (kindly note that all of the images used here are free images so no copyright issues.)



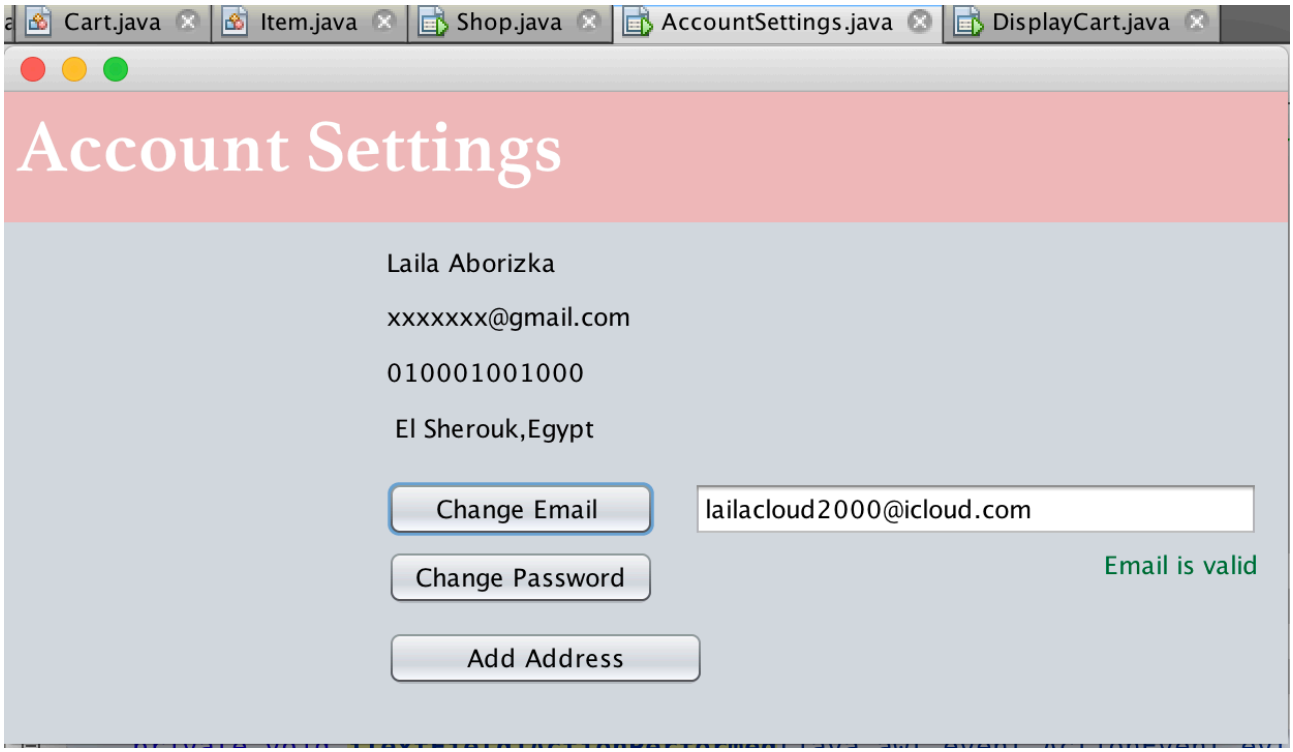
When the customer clicks the button Account Settings he/she is taken to the following interface



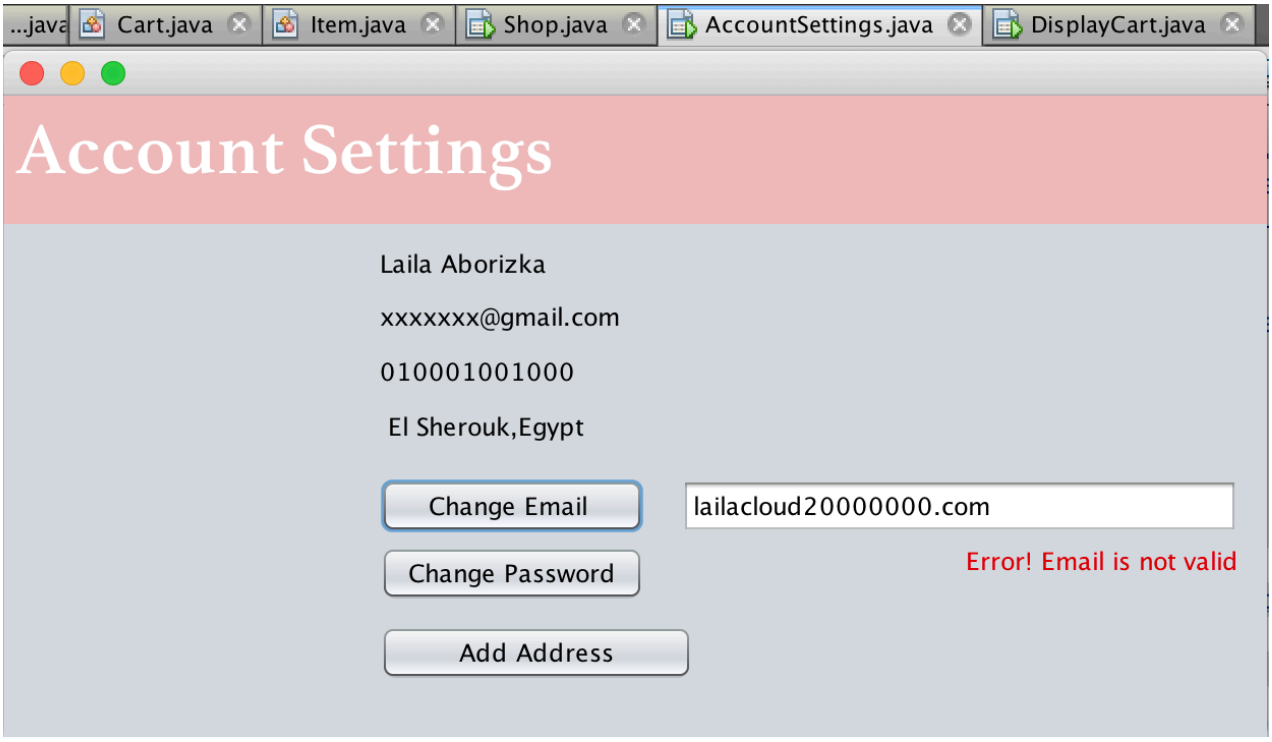
Then if the user decides to edit the details for example changing his email the window will look like this:



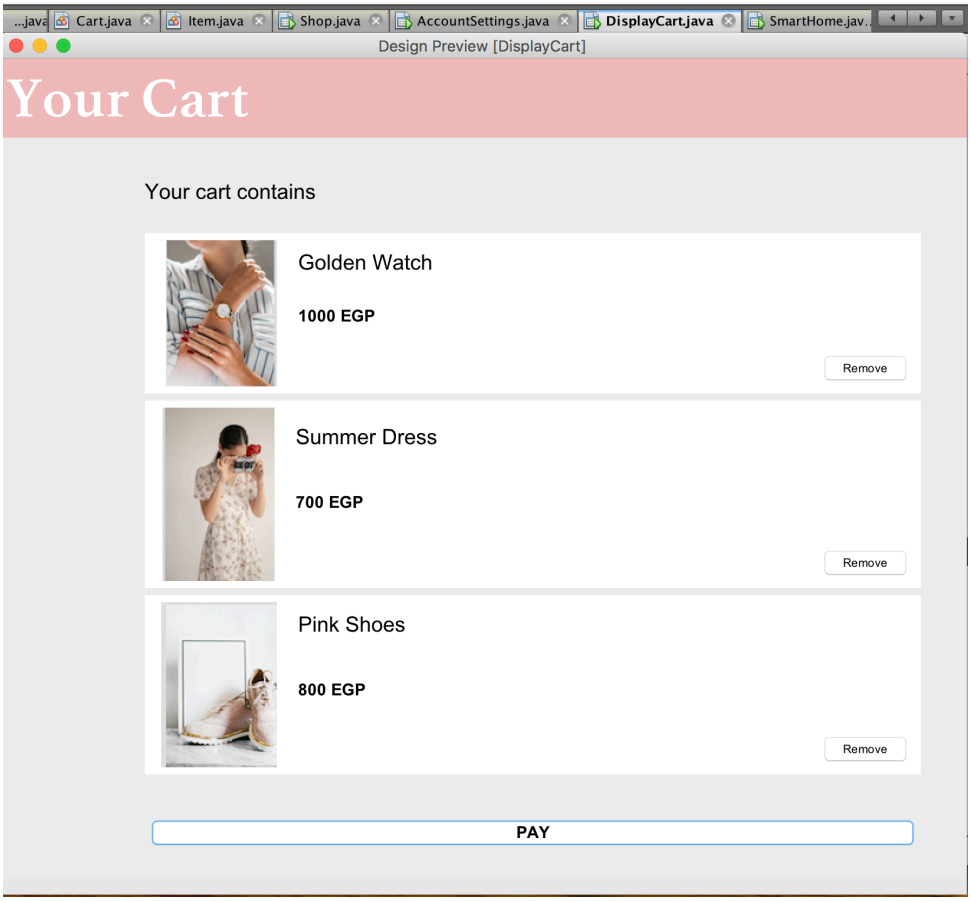
After inserting the new email the email is checked for validity :
IF VALID: the user gets a green message that says Email is valid



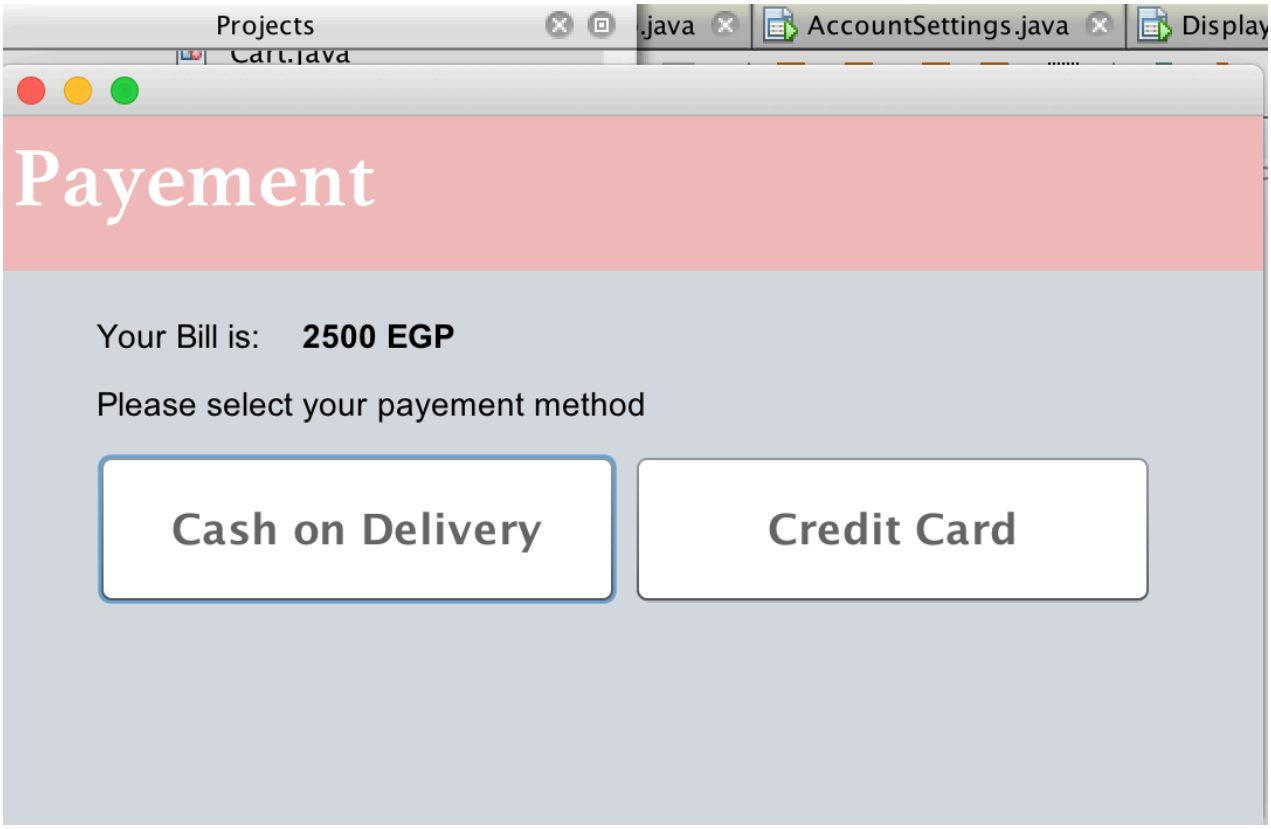
IF NOT VALID: the user gets an error message that says email is not valid!



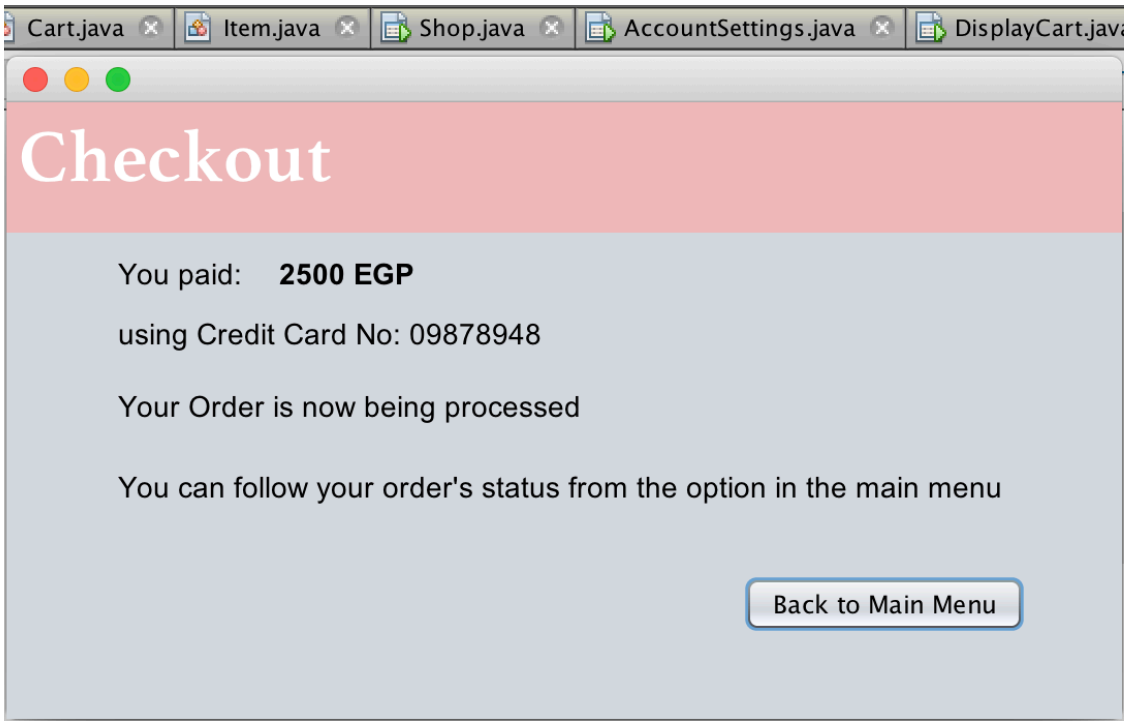
If the user chooses to display the cart from the main menu the following window will appear:



If the user chooses to pay and complete the shopping process the following window is displayed:



After selecting the payment method and inserting correct details that are validated by the system the user gets the Checkout window:



Section 3:

Several applications, ranging from high-performance servers to applications of an enterprise to Graphical User Interfaces as well as embedded systems, depend on the **event-based programming** paradigm.
Event-driven programming applies a programming idiom wherever programs use non-blocking Input/Output operations, and also the applied scientist breaks the computation into fine-grained event handlers that are related to the completion
This approach permits the interleaving of the many coincidental logical tasks with the lowest overhead, underneath the management of associate degree application-level cooperative hardware. every request executes some helpful work so either schedule more callbacks, contingent upon later events or invokes a continuation, that resumes the management flow of its logical caller.
The event-driven style has been demonstrated to realize high turnout in server applications, resource-constrained embedded devices, and business applications.

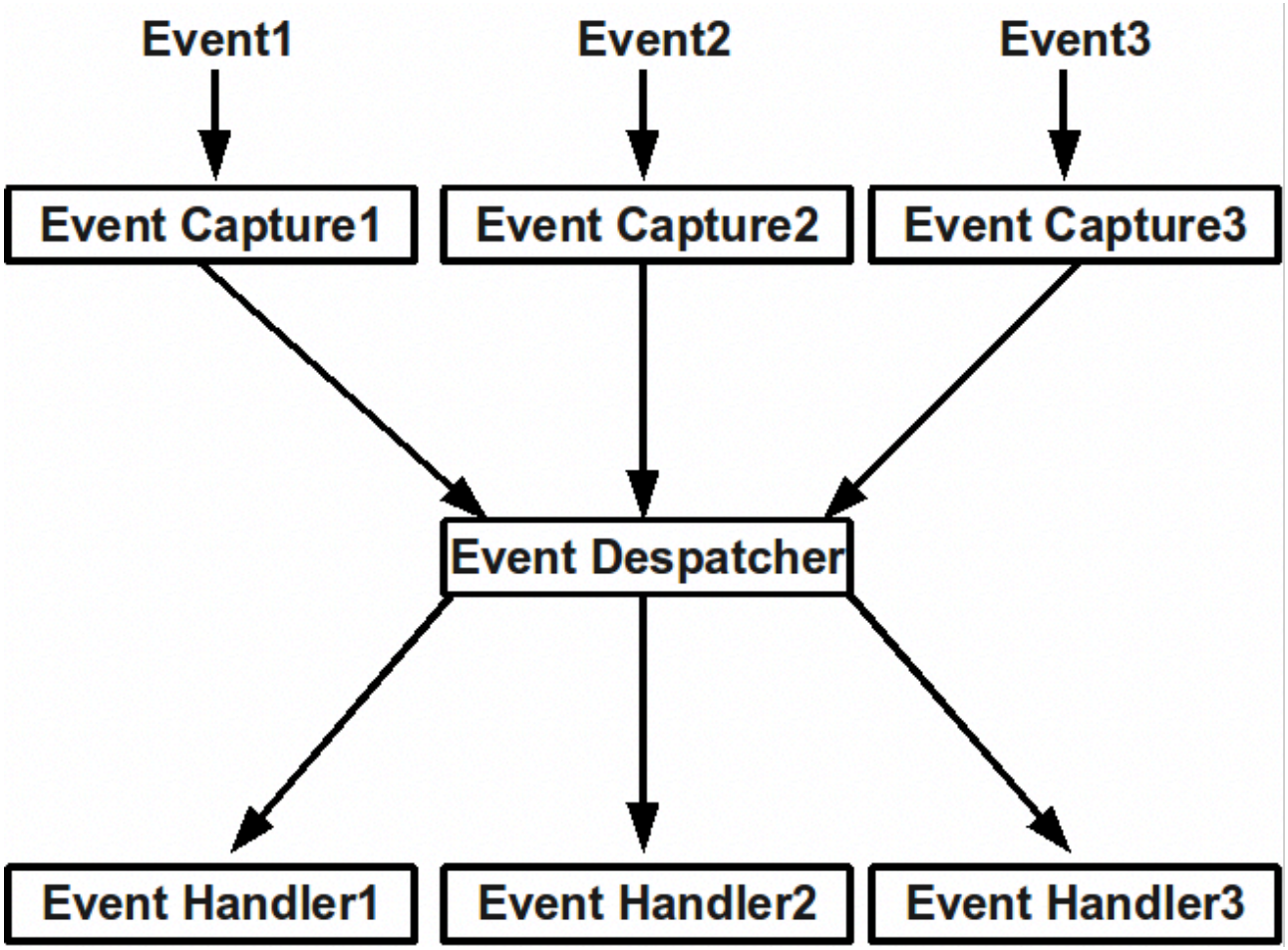
Event driven programming in servers:
The event-driven programming is often utilized in server programming in conjunction with non-blocking Input/Output. Non-blocking Input/Output libraries allow input/output operations to be regular in order that they won't not block within the software[2].

Event-driven programming in GUI:
Event-Driven Programming is appropriate for graphical user interface as a result of users naturally manufacture events like clicks, keypresses, mouse movements, drag drops, etc. it's as straightforward as that.Take games as an example. The command-line interface is like chess. The counterpart is looking forward to your move (input). once you make sure your move then he processes it and makes its move (output).However, graphical user interface is similar to real time strategy. Things are happening all the time. the game (GUI) isn't asleep, it lives in real-time and reacts to everything that's happening around. It reacts to events.

Event driven programming in embedded systems:
Most of the tiny embedded systems react to external or internal events somehow. The external event could be something like an interrupt, or change of signal level at Input/Output pins, a message packet coming from other system's parts through some interface [2].

How it works?
The events are generated when the user performs an action on a system. Examples of user actions are pressing a button or a key pad, touching a touch screen, clicking a mouse or moving it. The events could also result from sensors or devices connected to the system. Sometimes event may be internally generated, such as timeout event or a software exception. Regardless of the source or event type, the event driven programming is a programming paradigm in which the flow of the program is decided by events. The actual implementation of event driven programming have following sections of programs.

- Event Capture Section
- Event Despatch Section
- Event Handlers Section



1. Event Capture Section

This section of program captures the events, pre-processes and identifies the type of event.

2. Event Despatch Section

This section is responsible for mapping the events with the respective handler and calling the handler.

3. Event Handlers Section

This section's responsibility is implementing the activities; those should take place on occurrence of an event.

The majority of event driven programs can be described as state less, in other words when any application finishes processing an event, the application does not need to maintain its earlier event. When the event occurs, the respective handler is executed. So we can say that the execution flow is not dependent on the earlier events [3].

Concurrency

Concurrency can be defined as multiple computations happening at the same time. We can find Concurrency everywhere in modern programming:

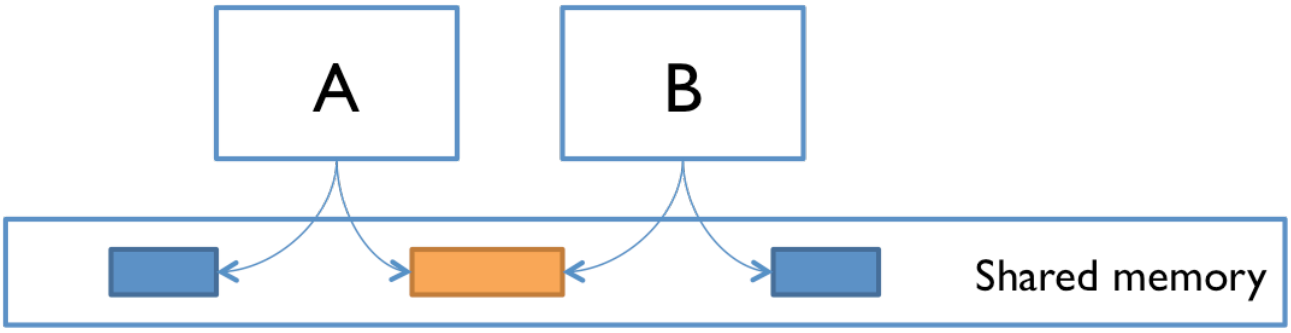
- Several computers in a network
- Several applications running on just one computer
- Multiple processors inside a computer (multiple processor cores on one chip)

In fact, concurrency is very important in modern programming:

- Web sites need to manage several users simultaneously.
- Mobile apps are required to do some of their processing on servers ("in the cloud").
- Graphical user interfaces most of the time require background work without interrupting the user. For example, Eclipse compiles Java code while the user is still editing it.

Processor clock speeds wont be increasing. Instead, we get more cores with each new generation of chips. So in the future, to get a computation to run faster, we have to divide a computation into concurrent pieces.

concurrent programming has two main models: shared memory and message passing.

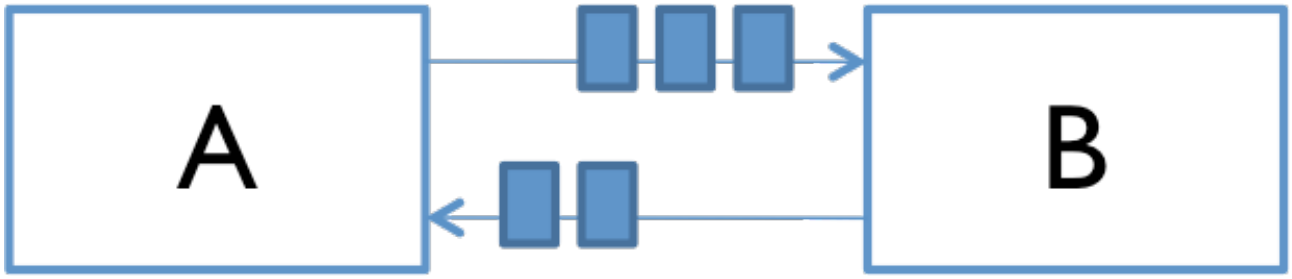


Shared memory:

In the shared memory model, concurrent modules interact through reading and writing objects in memory.

Other examples of this model:

- A and B may be two processors (or processor cores) within one computer, with the same physical memory.
- A and B may be two programs running on one computer, with a common filesystem.
- A and B may be two threads in the same Java program sharing same objects.

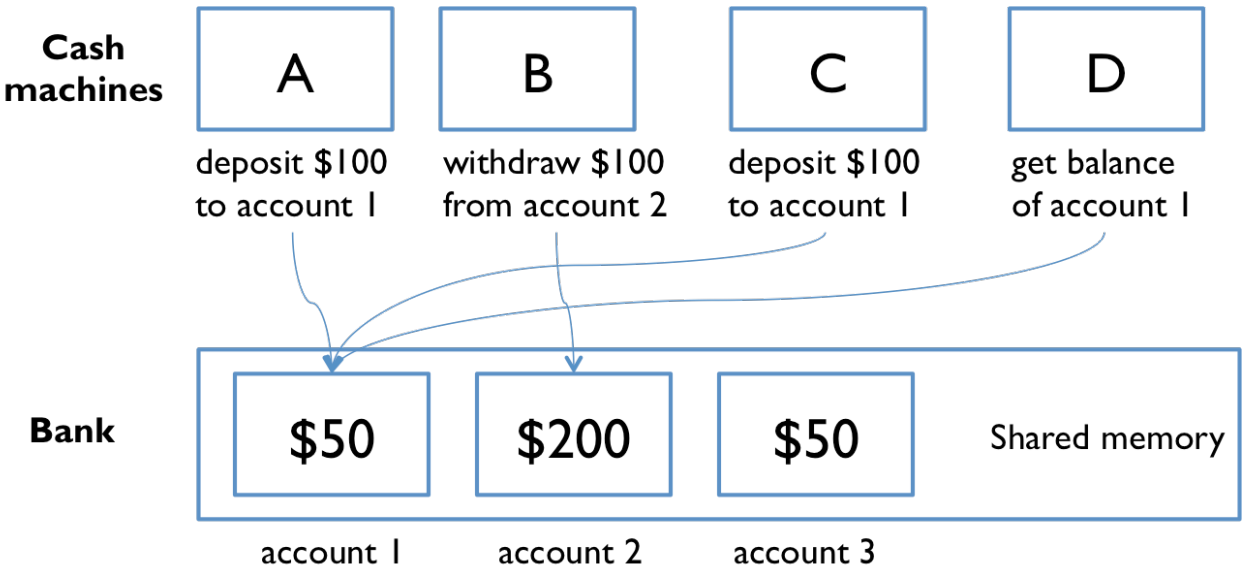


Message passing:
In this model, concurrent modules interact through sending messages to each other through a channel. Modules send off messages, and incoming messages to each module line up for handling. Examples for this model:

- A and B may be 2 computers in one network, communicating through network connections.
- A and B may be a web browser and a web server – A opens a connection to B, requests a web page, and B returns the web page data back to A.
- A and B might be an instant messaging client and server.

Shared Memory Example

A bank has cash machines that use a shared memory model, to allow all the cash machines to read and write the same account objects in memory [4].



b-

A **race condition** is a situation that is not desirable. It occurs when a device or system attempts to perform more than one operation at the same time, but due to the nature of the device or system, the operations should happen in the proper sequence to be done in a correct manner. To further explain, let us consider a simple however common **example**, a print spooler.

When a process aims to print a file, the file name is entered in a spooler directory. Another process, the printer daemon, checks to see if any files are present for printing, if any are found, it prints them and then removes the names from the directory.

The spooler directory has a large number of slots, numbered 0, 1, 2, ..., each can hold one file name. Also there are two shared variables, out: pointer to the next file to be printed, in: pointer to the next free slot in the directory.

At some point, slots zero to 3 are found to be empty (the files been printed) and slots 4 to 6 are full (containing names of files to be printed). More or less simultaneously, processes A and B decide they want to line a file for printing.

Process A reads and then stores 7, in a local variable called nextFreeSlot. After that a clock interrupt happens and the CPU decides that this is enough for process A and it switches to B.

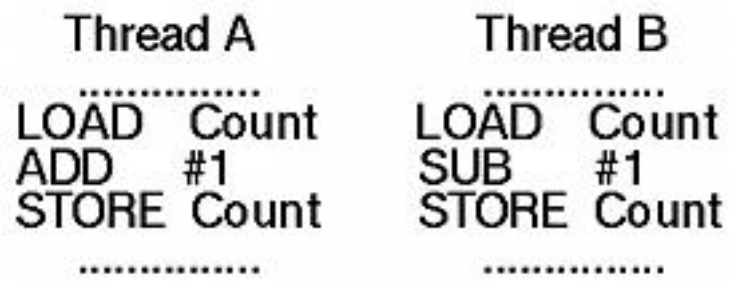
Process B will also read in, and will also get a 7, so it stores the name of its file in slot number 7 and then updates it to be 8.

After that, process A runs again, starting from where it left off. It looks at nextFreeSlot, finds a 7, and writes its file name in slot 7, doing that it erases the name that process B just added. Then it computes the next slot, 8, and sets in to 8. The spooler directory is now consistent, the printer daemon wont notice that something is wrong, however **process B will never receive any output.**

Another **Example** for Race Condition:

When more than one concurrently running threads/processes access a shared data item or resources and then the output is dependent on the order of execution, we get a race condition. Lets say we have two threads A and B.

Thread A increments the share variable count and Thread B decrements the count. If the current value of Count is 10, both execution orders will result in 10 because Count increases by 1 then decreases by 1 for the former case, and the opposite for the latter. However, if mutual exclusion is not there for protection, we might get difference results. Statements Count++ and Count-- might be translated as shown below:



The problem is while thread A is in the middle of executing Count++, the system may put A out and let B run. Same applies, while thread B is in the middle of executing Count--. If this happens, we get a problem. The following table shows the execution details. For every thread, the table shows the instruction executed and the content of register where it belongs. registers are part of a thread's environment and so if threads are different then the environment is different as well. As a result, the modification of a register by a thread only affects the thread itself and will not affect the registers of other threads. The last column shows the value of Count present in memory. Lets assume thread A is running. After thread A executes its LOAD instruction, a context switch turns thread B in. Thus, thread B executes its three instructions, resulting in the value of Count changing to 9. Then, the execution returns back to thread A, which continues with the remaining two instructions. Finally, the value of Count is changed to 11!

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
LOAD Count	10			10
		LOAD Count	10	10
		SUB #1	9	10
		STORE Count	9	9
ADD #1	11			10
STORE Count	11			11

This coming table shows the execution flow of executing thread B then thread A, and the result is 9!

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

This is, of course, a race condition. This race condition is due to no mutual exclusion.

Avoiding Race Conditions:

To avoid race condition we use Mutual Exclusion. It is a way of making sure that if one process is using a shared variable or file, the other processes will not be doing the same things.
The problem in the printer spooler occurs as a result of process B starting using one of the shared variables before process A finishing with it.
That part of the program within which the shared memory is accessed is called the critical section. If we are able to arrange the processes so that no two processes are ever in their critical regions at the same time, we can easily avoid race conditions.
However this alone is not sufficient for having parallel processes cooperate correctly and efficiently using shared data.
More tips for avoiding Race Condition:
1. Mutual Exclusion
2. No assumptions to be made concerning speeds or the number of CPUs.
3. No process running outside its critical region has the ability to block other processes.
4. No process should have to wait for a long time to enter its critical region.

REFERENCES:

[1]Balagurusamy E. Object Oriented Programming with C++, 6e. Tata McGraw-Hill Education; 2011 Jun 24.
[2]Fischer J, Majumdar R, Millstein T. Tasks: language support for event-driven programming. InProceedings of the 2007 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation 2007 Jan 15 (pp. 134-143).
[3]Dash NP, Dasgupta R, Chepada J, Halder A. Event driven programming for embedded systems-a finite state machine based approach. InProceeding of the 6th International Conference on Systems 2011 Jan.
[4]Saman Amarasinghe, Adam Chlipala, Srini Devadas, Michael Ernst, Max Goldman, John Guttag, Daniel Jackson, Rob Miller, Martin Rinard, and Armando Solar-Lezama, Fall 2014, MIT, Concurrency, https://web.mit.edu/6.005/www/fa14/classes/17-concurrency/.