CS244 Advanced Programming Applications

Dr Walid M. Aly

Lecture 2

Encapsulation

Schedule

W	Num	Date	Topics
W1	1	21 Feb	Revision
W2	2	28 Feb	Abstraction- Encapsulation-Access Modifiers-
W3	3	6 March	OO Relationship
W4	4	13 March	Polymorphism-(Assignment #1)
W5	5	20 March	Exceptions / java IO-(Assignment #2)
W6	6	27March	GUI
W7		3 April	W7 Assessment
W8	7	10 April	Events(1)
W9	8	17 April	Events(2
W10	9	24 April	Trip To Glasgow
W11	-	1 May	عيد العمال
W12	10	8 May	MVC Pattern (Assignment #3) W12 Assessment Threads
W13	11	15 May	JAVA.net- Assignment #4
W14	12	22 May	Enumerator-Collection framework
w15	---	29 May	Revision

Abstraction

- Abstractions is formed by reducing the information content of a concept typically to retain only information which is relevant for a particular purpose
- **Abstraction design** : **abstraction** reduce and factor out details so that one can focus on a few concepts.
- **Abstraction** allows the design stage to be separated from the implementation stage of the software development.

```
class Course{
String code;
int numOfStudents;
....
boolean isFull()
{....}
....
}
```

Example 1 : Abstraction of a Credit Card

```
public class  CreditCard{
double limit;
int number;
double balance;
static double final MAX_LIMIT=20000;
public CreditCard(int number){
this.number=number;
}

public void setLimit(double limit){
If(limit<=MAX_LIMIT)
this.limit=limit;
else  return;
}
public boolean buyWithCredit(double amount){
if ((balance+amount)>limit) return false;
balance=balance+amount;
return true;}
public void creditSettle(double amount){
balance=balance-amount;
}
}
```

Misuse of class

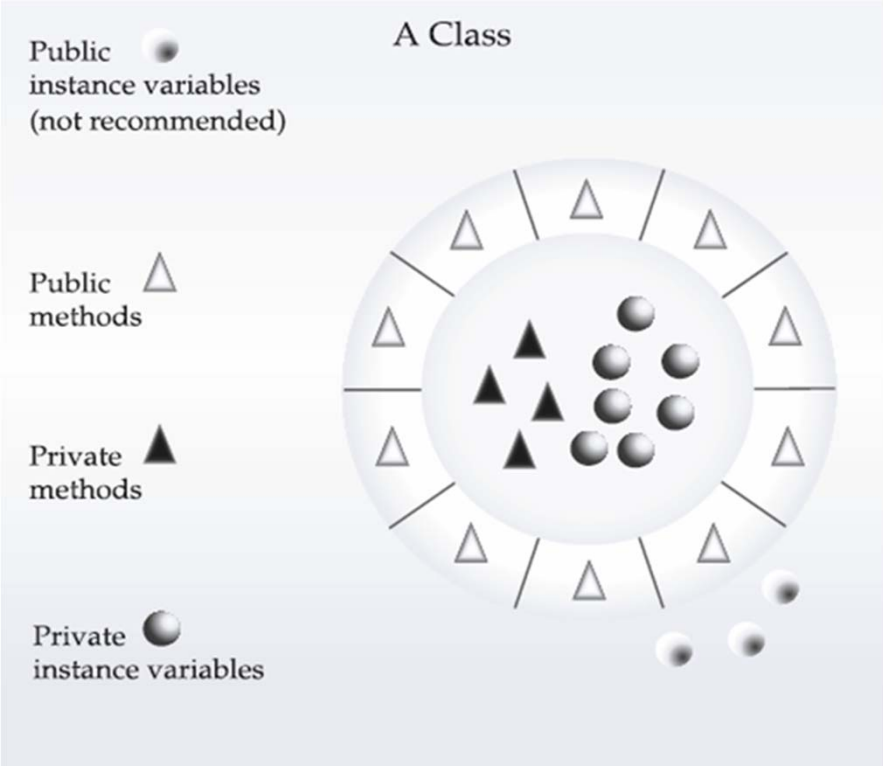
```
class Test{
public static void main (String [] arg){
CreditCard card1=new CreditCard(345);
card1.setLimit(5000);
card1.buyWithCredit(500);
card1.buyWithCredit(1500);
card1.limit=100000;
card1.buyWithCredit(15000);
}
}
```

Encapsulation

I. A language construct that facilitates the bundling of **data** with the **methods** operating on that data (this is achieved using the class structure)
II. A language mechanism for restricting access to some of the object's components, (this is achieved using access modifiers)

- A class should encapsulate only one idea , i.e. a class should be cohesive
- Your Encapsulation design should minimize dependency in order to ensure that there is a **loose-coupling**

```
class MP3Player
{
private int resistanceValue ;
public void raiseVolume()
{……….……}
}
```



Better Encapsulation of class CreditCard

```
public class  CreditCard{
 private double limit;
 private  int number;
 private  double balance;
public static double fimal MAX_LIMIT=2000;
public CreditCard(int number){
this.number=number;
}
public void setLimit(double limit){
If(limit<=MAX_LIMIT)
this.limit=limit;
else  return;
}
public boolean buyWithCredit(double amount){
if ((balance+amount)>limit) return false;
balance=balance+amount;
return true;
}
public void creditSettle(double amount){
balance=balance-amount;
}}
```

```
class Test{
public static void main (String [] arg){
CreditCard card1=new CreditCard(345);
card1.setLimit(5000);
card1.buyWithCredit(500);
card1.buyWithCredit(1500);
card1.limit=100000;
card1.buyWithCredit(15000);
}
}
```

Access Control

Java provides several modifiers that control access to data fields, methods, and classes

- **public makes classes, methods, and data fields accessible from any class.**
- **private makes methods and data fields accessible only from within its own class.**
- If **public or private is not used, then by default the classes, methods, and data fields are accessible by** any class in the same package. This is known as ***package-access.***
- If protected **is used,** then **that** member can be accessed by own class and classes from same package and from subclass outside the package .

```
public int i;
private double j;
public void m (){}
int x;
```

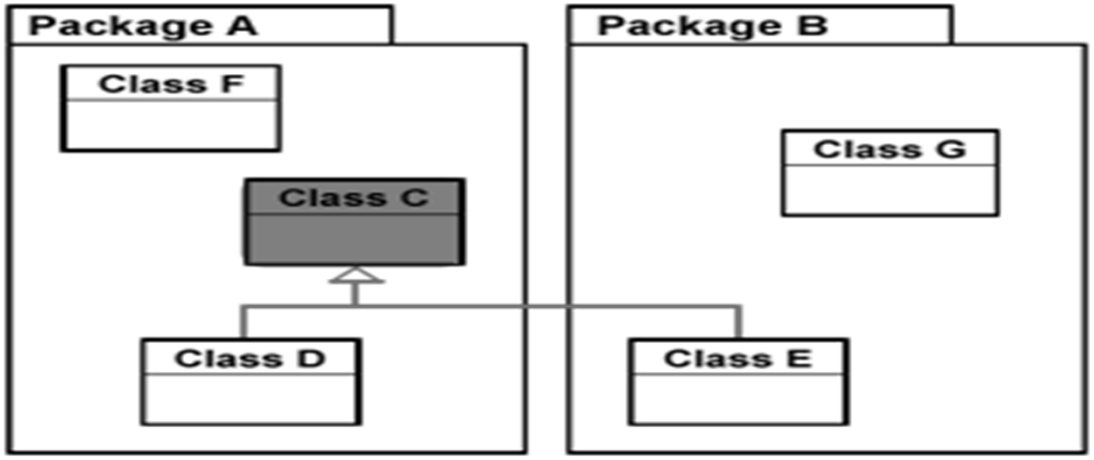
The default constructor has the same access as its class

Local Variables can not have an access modifier

The Different Levels of Access Control

Visibility	public	protected	default	private
From the same class	Yes	Yes	Yes	Yes
From any class in the same package	Yes	Yes	Yes	No
From any class outside the package	Yes	No	No	No
From a subclass in the same package	Yes	Yes	Yes	No
From a subclass outside the same package	Yes	Yes	No	No

Access modifiers apply to class members (variables and methods) and **constructors** too.



Class D,E subclasses of class C

Data Field Encapsulation

- Instance variables are declared private to prevent misuse.
- providing methods that can be used to read/write the state rather than accessing the state directly.

```
Person p1=new Person();
p1.age=10;
System.out.println(p1.age);
```

```
Person p1=new Person();
p1.setAge(10);
```

```
public class Person{
private int age;
public void setAge(int age ){
if (age<0)
{
System.out.println("unaccepted value");
return;
}
this.age=age;
}
public int getAge(){
return age;
}
}
```

Data Field Encapsulation

- Instance variables are declared private to prevent misuse.
- providing methods that can be used to read/write the state rather than accessing the state directly.

Accessors and mutators

- **Accessor method (getters):** a method that provides access to the state of an object to be accessed.

A **get method signature:**

public returnType get*PropertyName()*

If the **returnType is boolean :**

public boolean is*PropertyName()*

- Mutator method(setters): a method that modifies an object's state.

A set method signature:

public void set*PropertyName(dataType propertyValue)*

```
public class Person{
private int age;
private String name;
private boolean adult;
public int getAge()
{return age}
public void setAge(int age )
{this.age.age;}
public String getName()
{return name;}
public void setName(String name)
{this.name=name;}
public boolean isAdult()
{return adult}
public void setAdult(boolean adult)
{this.adult=adult;}

}
```

Example :Class **GasTank**

- Write a class named GasTank containing:
  - An instance variable named amount of type double, initialized to 0.
  - A method named addGas that accepts a parameter of type double . The value of the amount instance variable is increased by the value of the parameter.
  - A method named useGas that accepts a parameter of type double . The value of the amount instance variable is decreased by the value of the parameter.
  - A method named getGasLevel that accepts no parameters. getGasLevel returns the value of the amount instance variable.

```
public class GasTank {
 private double amount = 0;
 public void addGas ( double doubleAmount)
 {
 amount += doubleAmount;
}
public void useGas ( double doubleAmount)
{
amount -= doubleAmount; }
public double getGasLevel () {
return amount;
 }
}
```

Example 2 :Class **MusicAlbum**

Implement Class **MusicAlbum** which encapsulated a music Album, each album has a string variable ***albumTitle*** and a String variable ***singer*** representing the name of singer, double variable ***price*** representing the price of album, objects of this class are Initialized using all of its instance variables.

The class has **accessor** methods for all its Variables and a **mutator** method for price

```
public class MusicAlbum {
private String albumTitle;
private String  singer;
private double price;

Public MusicAlbum(String albumTitle, String singer, int price)
{
 this.albumTitle=albumTitle;
 this.singer=singer;
 this.price=price;
}

public String getAlbumTitle()
{
 return albumTitle;
}

public String getSinger()
{
 return singer;
}
```

```
public double getPrice()
{
 return price;
}

public void setPrice ( double price)
{
            this .price=price;
}
}
```

Example 3 :Class **Stack**

Design a java class that encapsulate the data structure Stack (Last in First out).

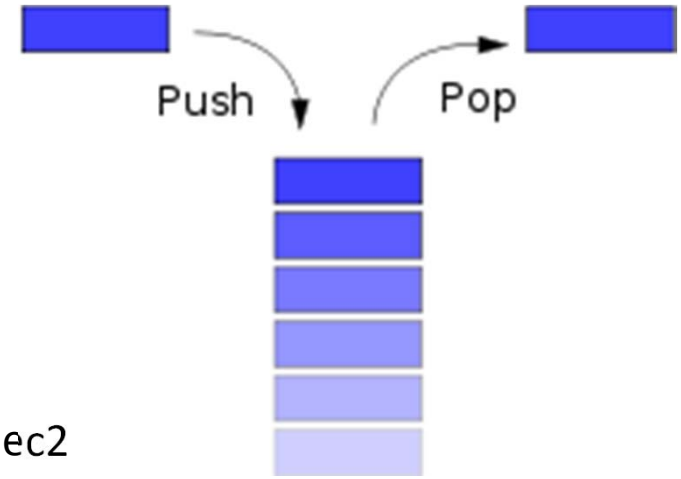
- The class has 2 methods

(push) : method for adding items to stack, the push operation adds items to the top of the list

(pop) :method for retrieving items from the stack, . The pop operation removes an item from the top of the list, and returns its value to the caller.

- in the case of overflow the user should be informed with a message
- in the case of underflow, the user should be informed with a message & a value of zero is returned.

Assumptions : The stack will hold up to 10 +ve integer values.



Class Stack

```
/* This class defines an integer stack
that can hold 10 values*/
class Stack {
  private int stck[] = new int[10];
  private int tos;

Stack() {
    tos = -1;
  }

  // Push an item onto the stack
 public void push(int item) {
    if(tos==9)
      System.out.println("Stack is full.");
    else
      stck[++tos] = item;
  }
```

```
  // Pop an item from the stack
public int pop() {
    if(tos < 0) {
      System.out.println("Stack underflow.");
      return 0;
    }
    else
      return stck[tos--];
  }
}
```

```
class TestStack {
  public static void main(String args[]) {
    int element;
    Stack mystack1 = new Stack();

    // push some numbers onto the stack
     mystack1.push(1);
     mystack1.push(17);
    //mystack1.stck[3]=4  compile error call is encapsulated stck is private
     mystack1.push(20);
    // pop some numbers off the stack
       element=mystack1.pop();
       System.out.println("Element is "+element);
       element=mystack1.pop();
       System.out.println("Element is "+element);
  }
}
```

```
Element is 20
Element is 17
Press any key to continue . . .
```

Example 3 :Class **Car**

- Design and implement a class called Car that contains instance data that represents the make, model, and year of the car (as a String, String and int values respectively). Define the Car constructor to initialize these values (in that order).
- Include getter and setter methods for all instance data

```
public class Car {
        private String make, model;
        private int year;
        public Car(String make, String model, int year) {
                setMake(make);
                setModel(model);
                setYear(year);
        }
        public String getMake() {return make;}
        public void setMake(String make) {this.make = make;}
        public String getModel() {return model;}
        public void setModel(String model) {this. model = model;}
        public int getYear() {return year;}
         public void setYear(int year){this.year=year;}
}
```

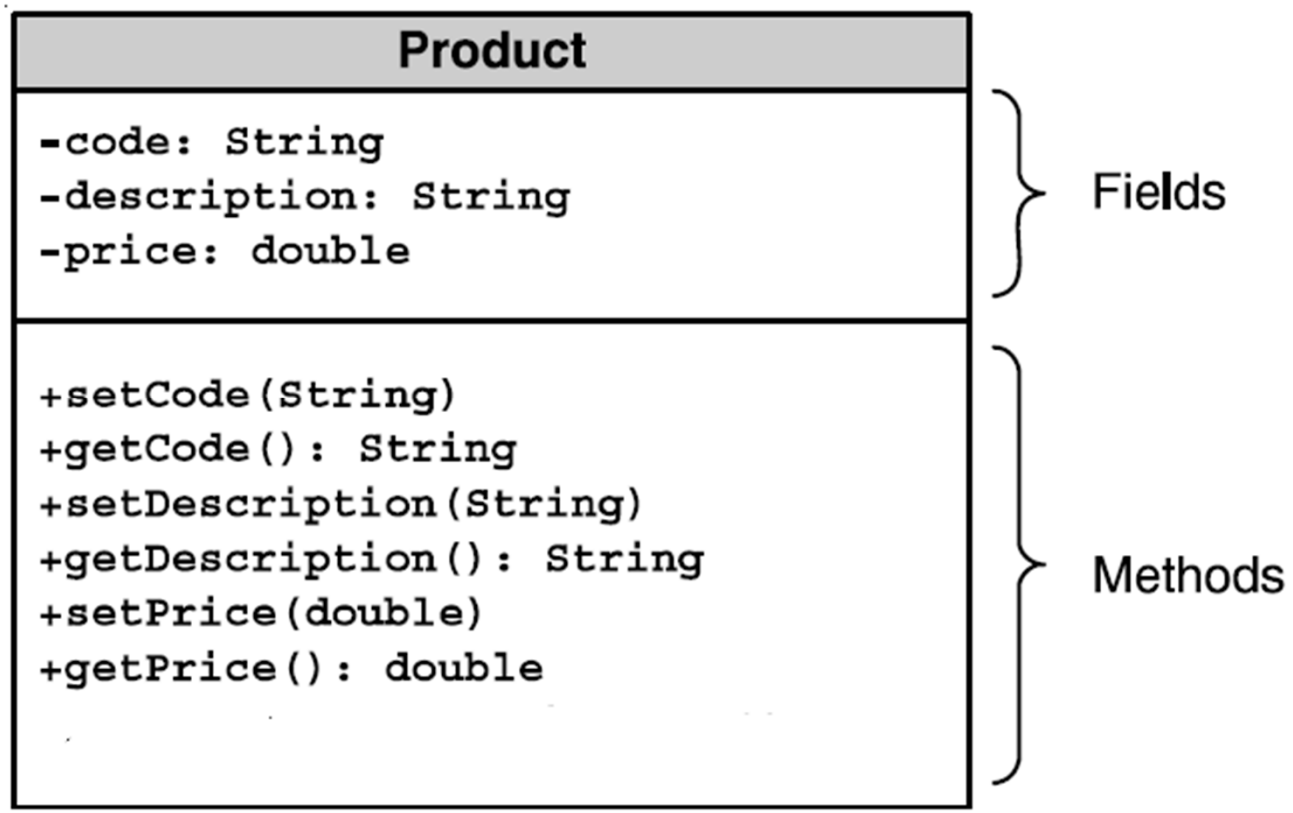
Example 4 : Abstraction of a Farm Animal

```
public class Animal
{
    private Location loc;
    private double energyReserves;

    public boolean isHungry() {
        return energyReserves < 2.5;
    }
    public void eat(Food f) {
        // Consume food
        energyReserves += f.getCalories();
    }
    public void moveTo(Location l) {
        // Move to new location
        loc = l;
    }
}
```

```
class Food
{
private double calories
public double getCalaories()
{
return calories;
}
……
}
```

```
class Location
{
…….
}
```

Example 5 : Abstraction of a Product



-: private

-+:public

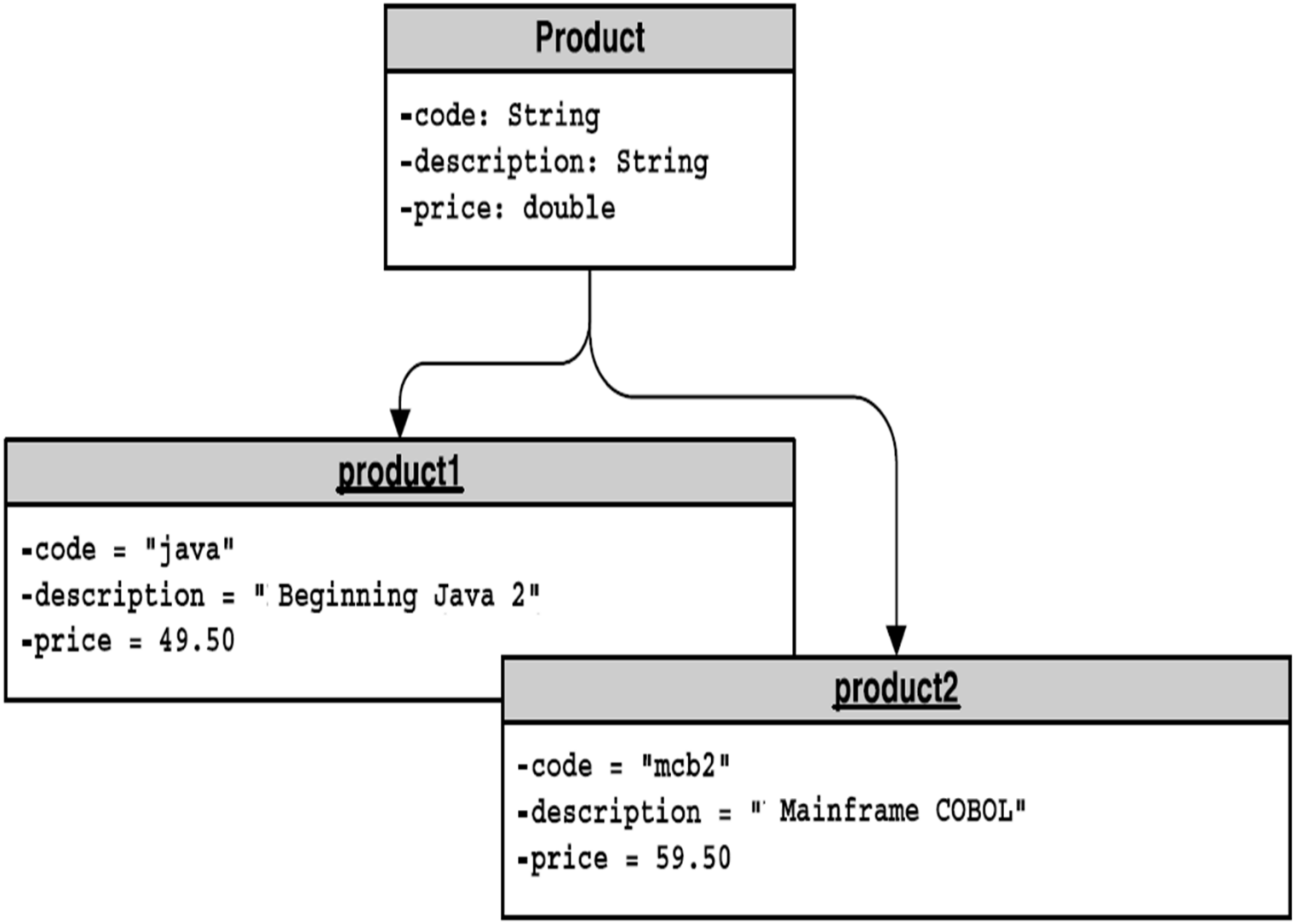
- Add a default empty no-arg constructor
- Add a constructor for instance variable initialization using arguments

```
public class Product
{
        private String code;
        private String description;
        private double price;

        public Product()
        {
                code = "";
                description = "";
                price = 0;
        }

        public Product(String code, String description, double price)
        {
                this.code = code;
                this.description = description;
                this.price = price;
        }
```

```
public void setCode(String code)
{
this.code = code;
}

public String getCode(){
return code;
}

public void setDescription(String description)
{
this.description = description;
}

public String getDescription()
{
return description;
}
```

```
public void setPrice(double price)
{
this.price = price;
}

public double getPrice()
{
return price;
}

}
```

Creating Object of class Product



```
class Test{
public static void main (String [] arg){
Product product1=new Product();
product1.setCode("Java");
product1.setDescription("Begining Java 2");
product1.setPrice(49.5 );
Product product2=new Product("mcb2","MainFrame Cobol",59.50);
displayProduct(product1);
displayProduct(product2);
}
public static void displayProduct(Product product)
{
 String productCode=product.getCode();
 String productDescription=product.getDescription();
 double productPrice=product.getPrice();
System.out.println("Product code is " +productCode +  ", description
:"+ productDescription + ", price "+productPrice);
}
}
```

```
Product code is Java description :Begining Java 2 price 49.5
Product code is mcb2 description :MainFrame Cobol price 59.5
```

Example 6 : Abstraction of a MobilePhone