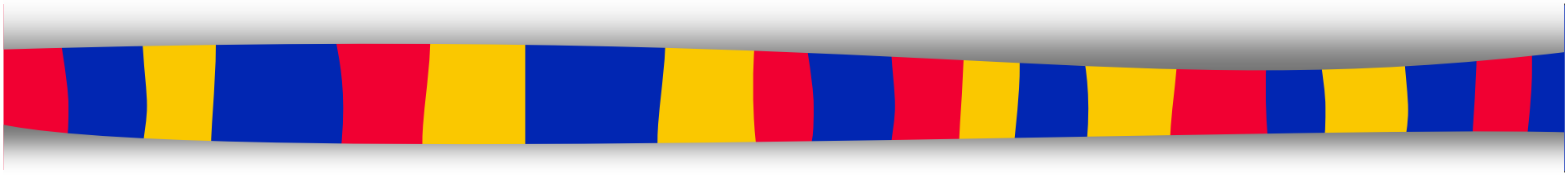
COMP-248
Object Oriented Programming I



Classes and Objects

By Emad Shihab, PhD, Fall 2015,
Parts of the slides are taken from Prof. L. Kosseim
Adapted for Section EE by S. Ghaderpanah, Fall 2015

Next:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. **Some notions of OOP**
3. Passing and returning objects
4. Recap

2- Some notions of OOP

- **Information hiding (aka Abstraction)**
  - separating how to use a class from how it is implemented
  - Useful since a programmer who uses your class need not be overloaded with implantation details. All they need to know is **how to use your class**

2- Some notions of OOP

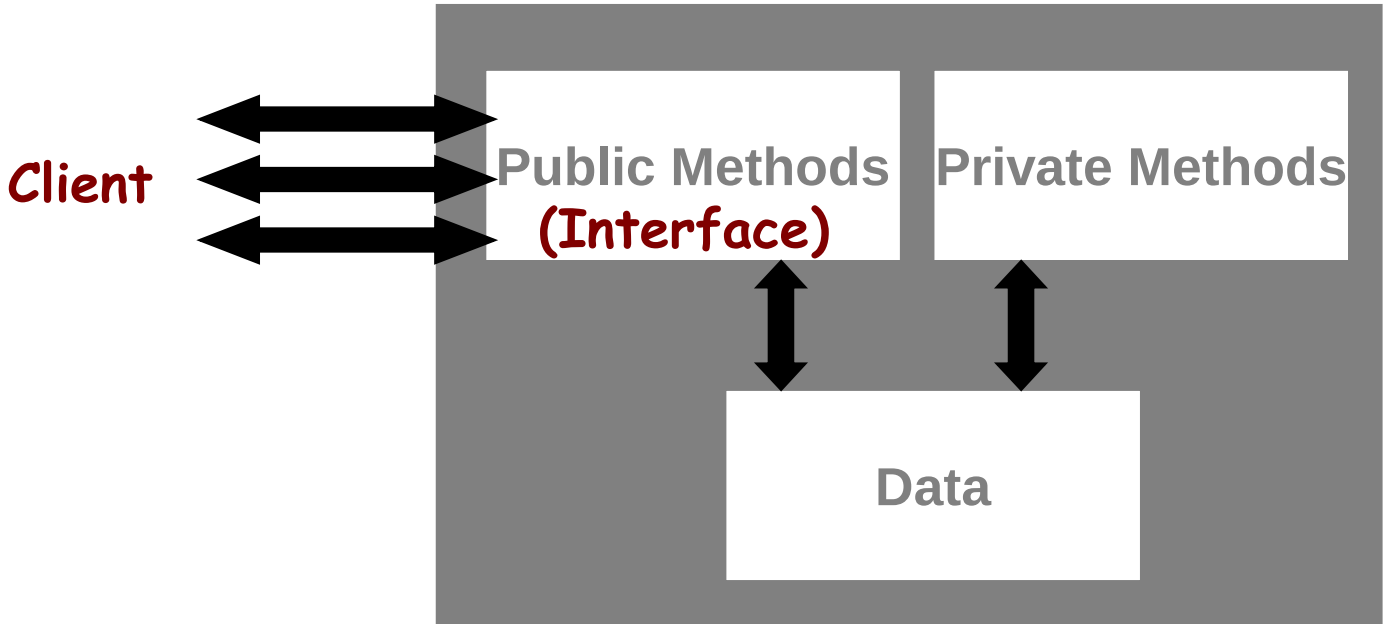
- **Encapsulation**
  - The data and methods of a class are combined into a single unit (i.e., a class object), which hides the implementation details
  - Knowing the details is unnecessary because interaction with the object occurs via a well-defined and simple interface
  - In Java, hiding details is done by marking them `private`

2- Some notions of OOP

- **Interface**
  - An object interacts with the rest of the program via an interface
  - Interface = set of methods that allow access to the object
  - The interface hides how a method is implemented

Encapsulation

- An encapsulated object can be thought of as a *black box*
  - its inner workings are hidden to the client
  - the client can access the object only by invoking the methods of the interface



Visibility Modifiers

Java has 4 visibility modifiers:

- **`public`**
- **`private`**
- **`protected`** (involves inheritance, see COMP 249)
- **`default or package`** (COMP 249)

can be applied to all members (data and methods)

Visibility modifiers

- **public** members
  - can be directly accessed from anywhere (inside & outside the object)
  - violate encapsulation
- **private** members
  - can only be accessed from inside the class definition
- by default…
  - members can be accessed by any class in the same package
  - i.e. access is more open than private, but more strict than public

Visibility modifiers

- **Data**
  - should be **private**
  - public data violate encapsulation
  - constants are OK (but not encouraged) to be public because they cannot be modified anyways
- Methods:
  - should mostly be **public**
    - if the method provides the object's services so that it can be invoked by clients also called ***service method***
  - should be **private**
    - if the method is created simply to assist a service method

    also called a ***support method***

Visibility modifiers

	`public`	`private`
Variables	NO! Violates encapsulation	YES! Enforces encapsulation
Methods	Yes, if method provides services to clients	Yes, if method supports other methods in the class

Example

```
public class Account {
  private double rate;
  private long acctNumber;
  private double balance;
  private String name;
  public Account(String owner, long nb,
    double init) {
     name = owner;
     acctNumber = nb;
     balance = init;
     rate = 0.02;
  }
  public double deposit(double amount)   {
     if (amount < 0)  // deposit is negative
       System.out.println("Error");
     else
        balance += amount;
     return balance;
  }
  private void changeRate(double newRate) {
     rate = newRate;
  }
```

Account.java

```
public class Banking
{
  public static void main (String[] args)
  {
    Account acct1;
    double tedBal;

    acct1 = new Account("Ted", 72, 102.56);

    acct1.deposit(25.85); // 3. OK?

    acct1.balance = 0;  // 4. OK?

    tedBal = acct1.deposit(500.00); //5. OK?

    acct1.changeRate(5.5); // 6. OK?
  }
}
```

driver

Example

```
public class Printer
{
  private final int DEFAULT_NB = 10;

  public void printMany(int nbTimes, String theMessage)
  {
    if (nbTimes < 0 || nbTimes > 100)
        internalUseOnly(DEFAULT_NB, theMessage);
    else
       internalUseOnly(nbTimes, theMessage);
  }

  private void internalUseOnly(int nbTimes, String AMess)
  {
    for (int i = 1; i <= nbTimes; i++)
       System.out.println(AMess);
  }
}
```

class def.

```
public class PrinterDriver
{
  public static void main (String[] args)
  {
    Printer myPrinter = new Printer();

    myPrinter.printMany(5, "hello");  // 1. OK?
    System.out.println(myPrinter.DEFAULT_NB); // 2. OK?
    myPrinter.internalUseOnly(5, "hello"); // 3. OK?
  }
}
```

driver

Accessor and Mutator Methods

data members are usually private

so to access them, we usually have *set* and *get* methods

mutator (setX): sets the data X and makes sure it stays in a coherent state

accessor (getX): returns the value of data X

```
public class Time {
  private int hour;
  private int minutes;
  private int seconds;

  public void setHour(_____________________) {
    ____________________________________________;
  }

  public void setMinute(_____________________) {
    ____________________________________________;
  }

  public void setSecond(_____________________) {
    ____________________________________________;
  }
```

```
  public int getHour(_____________________){
     ____________________________________________;
  }
  public int getMinute(_____________________){
     ____________________________________________;
  }
  public int getSecond(_____________________){
     ____________________________________________;
  }
  // constructor
  public Time(int h, int m, int s) {
     ____________________________________________;
     ____________________________________________;
     ____________________________________________;
   }
}
```

why should the *set* and *get* methods be public?

Mutators that return a boolean

if a mutator is given a value that would make the object in an **invalid state**, it can:

1. **display an error message**, and quit the program, OR
2. **return a boolean** (false), and let the calling method decide what to do

```
public void setHour(int newHour)
{
   if (newHour > 24 || newHour < 0)
   {
      System.out.println("Error.");
      System.exit(0);
    }
    else
      hour = newHour;
}
```

```
Public boolean setHour(int newHour)
{
   if (newHour > 24 || newHour < 0)
   {
      return false
    }

   hour = newHour;
   return true;

}
```

Next:

1. Writing our own classes
   1.1 Objects vs classes
   1.2 Instance Variables
   1.3 Methods
2. Some notions of OOP
3. **Passing and returning objects**
4. Recap

3- Passing and returning objects

An object can be a parameter to a method

ex:

```
public int compareTo(String AMess)
  {…}
```

String class

```
String originalString = "hello";
int result = originalString.compareTo("bye bye");
```

```
public void printMany(int nbTimes, String AMess)
  {…}
```

MyPrinters class

```
MyPrinter p = new MyPrinter();
p.printMany(10, "hello");
```

Example: compare 2 Time objects

```
public class Time {
  private int hour;
  private int minutes;
  private int seconds;

  public Time(int h, int m, int s) {…}
  public void setHour(int h)  {…}
  …
  public int getSecond(){…}

// method to check if a time < another time
  Public boolean lessThanTime (Time anotherTime)
  {
    if (hour < anotherTime.getHour())
       return true;
    else if (hour > anotherTime.getHour())
       return false;
    else if (minutes < anotherTime.getMinute())
       return true;
    else if (minutes > anotherTime.getMinute())
       return false;
    return (seconds < anotherTime.getSecond());
   }
…
```

Time.java

```
public class TimeDriver
{
  public static void main (String[] args)
  {
    // declare 2 time objects
      Time time1 = new Time(5,4,1);
      Time time2 = new Time(1,1,2);


    // check if time1 < time2
    if (time1.lessThanTime (time2))
     System.out.println("ok");
  }
}
```

driver

Passing and returning objects

An object can be returned by a method

ex:

```
public String replace(char oldChar, char newChar)
  {…}
```

String class

```
String myString = "hello";
String newString = myString.replace('l', 'm');
```

```
public String toString()
   {…}
```

MyPrinters class

```
MyPrinters p = new MyPrinters();
System.out.print(p.toString());
System.out.print(p);
```

Example: Add 2 Time Objects

```
public class Time {
  private int hour;
  private int minutes;
  private int seconds;

  …
  // add a time to the current time
  public  Time  add( Time t2 )
  {
    Time result = new Time(0,0,0);
    result.hour = hour + t2.getHour();
    result.minute = minutes + t2.getMinute();
    result.second = seconds + t2.getSecond();
    if (result.second >= 60) {
          result.second -= 60;
          result.minute += 1;
    }
    if (result.minute >= 60) {
          result.minute -= 60;
          result.hour += 1;
    }
    return result;
   }
…
```

Time.java

```
public class TimeDriver
{
  public static void main (String[] args)
  {
     Time t1 = new Time(9,10,50);
     Time t2 = new Time(19,5,5);
     Time t3 = new Time(0,0,0);

     // t3 is the sum of t1 and t2
     t3 = t1.add(t2);  ;

  }
}
```

driver

Next:

1. Writing our own classes
   - 1.1 Objects vs classes
   - 1.2 Instance Variables
   - 1.3 Methods
2. Some notions of OOP
3. Passing and returning objects
4. **Recap**

4- Recap

- To define a class:
  - **data members** (attributes)

    declared outside any method

    must decide:

    visibility (private? public?)
  - **methods**

    must decide:

    visibility (private? public?)

    type of result (void?, int?, boolean?…)

    number and types of parameters

    actual code of the method

    static or not? (so far… never static)
- 2 files:

  the class definition (ex. Coin.java, BankAccount.java)

  the driver program (ex. CoinFlip.java, Banking.java)

Just checking …

A set method is:

A. an accessor method
B. a mutator method
C. a recursive method
D. none of the above

Just checking …

Accessor methods:

A. return the value of an instance variable
B. promotes abstraction
C. both A and B
D. none of the above

Just checking …

`this` refers to:

A. instance variables
B. local variables
C. global variables
D. the calling object

Just checking …

The name of a method and the list of _____ types in the heading of the method definition is called the method signature.

A. parameter
B. local variable
C. return
D. primitive

Example: Pet Class

```
public class Pet
{
    private String name;
    private int age;//in years
    private double weight;//in pounds

    // constructor
    public Pet(String initialName, int initialAge,
                                        double initialWeight)
    {
        name = initialName;
        if ((initialAge < 0) || (initialWeight < 0))
        {
            System.out.println("Error: Negative age or weight.");
            System.exit(0);
        }
        else
        {
            age = initialAge;
            weight = initialWeight;
        }
    }

    // another constructor
    public Pet(String initialName)
    {
        name = initialName;
        age = 0;
        weight = 0;
    }
```

Example: Pet Constructors

```
…
    // another constructor
    public Pet(int initialAge)
    {
        name = "No name yet.";
        weight = 0;
        if (initialAge < 0)
        {
            System.out.println("Error: Negative age.");
            System.exit(0);
        }
        else
            age = initialAge;
    }

    // another constructor
    public Pet(double initialWeight)
    {
        name = "No name yet";
        age = 0;
        if (initialWeight < 0)
        {
            System.out.println("Error: Negative weight.");
            System.exit(0);
        }
        else
            weight = initialWeight;
    }
```

Example: Pet Mutators

```
…
    // Mutator for all attributes
    public void set(String newName, int newAge, double
    newWeight)
    {
        name = newName;
        if ((newAge < 0) || (newWeight < 0))
        {
     System.out.println("Error: Negative age or weight.");
            System.exit(0);
       }
       else
        {
            age = newAge;
           weight = newWeight;
       }
    }

    // mutator for the name attribute
    public void setName(String newName)
    {
        name = newName;
    }
```

Example: Pet Mutators

```
…
    // mutator for the age attribute
    public void setAge(int newAge)
    {
        if (newAge < 0)
        {
            System.out.println("Error: Negative age.");
            System.exit(0);
        }
        else
            age = newAge;
    }

    // mutator for the weight attribute
   public void setWeight(double newWeight)
    {
        if (newWeight < 0)
        {
           System.out.println("Error: Negative weight.");
           System.exit(0);
        }
        else
            weight = newWeight;
    }
```

Example: Pet Accessors

```
…
    // accessor for the name attribute
    public String getName( )
    {
        return name;
    }

    // accessor for the age attribute
    public int getAge( )
    {
        return age;
    }

    // accessor for the weight attribute
    public double getWeight( )
    {
        return weight;
    }
```

Example: PetDemo

```
public class PetDemo
{
    public static void main(String[] args)
    {
        // let's create a pet object
        Pet usersPet = new Pet("Jane Doe");
        System.out.println("My records on your pet are incomplete.");
        System.out.println("Here is what they currently say:");
        System.out.println(usersPet); // this calls the toString method


        Scanner keyboard = new Scanner(System.in);
        System.out.println("Please enter the pet's name:");
        String name = keyboard.nextLine( );
        System.out.println("Please enter the pet's age:");
        int age = keyboard.nextInt( );
        System.out.println("Please enter the pet's weight:");
        double weight = keyboard.nextDouble( );
        usersPet.set(name, age, weight);
        System.out.println("My records now say:");
        System.out.println(usersPet);
    }
}
```

Next, we will see:

Arrays of Objects