Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

2022 Autumn – CSE213 - ScratchPad (for ALL sections)

Q: What programming language have you used in CSC101 & CSE203?

A:

Section	CSC101	CSE203
1	Mostly: C++	Mostly: C++
2	Mostly: C++	Mostly: C++
3	Mostly: C++	Mostly: C++
4	Mostly: C++	Mostly: C++

Q: Assume you are in CSC101, How would you implement a program to populate and show details of a student info such as id, name, cgpa, etc.?

A: ?????

- By declaring 3 arrays of ints, strings and floats for ids, names & cgpas for n students
- Problem: mapping among id, name & cgpa is loosely coupled and vulnerable
- Better choice: class

Q: What is an array?

- Its a collection
- Its a data structure
- Homogeneous (same type) collection of elements (data/??/??);

Q: What is 'class'?

- Its a collection
- Its a user defined **type**
- Heterogeneous (can be different type) collection of elements;

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Q: What is the scope of a variable?

A: Scope of a memory (represented by variables / pointers) is the territory from where the memory can be accessed. except global scope, a local scope It is represented by a block{..}

- A local scope can be represented by:
  - block associated with function/method definition/body
  - block associated with loops
  - conditional constructs such as if/else/switch-case
  - struct/class definition/body
  - namespace
  - enumerated type
  - bare-block
- we can't use same name to declare a memory twice in same scope
- If a variable/object is declared outside of a block, then we can say the scope is **global**
  - Global scope is not allowed in many programming languages including java

---

Q: What is the scope of a class/struct member (field or method)?

Scope of a class/struct member is dictated whether the member is accessible from outside the class/struct or not.

- By assessing a field (member-data) means the data can be read-from or write-into the field from a function/method who is trying to access the field
- By assessing a method (member-function) means the method can be called by the caller function/method who is trying to call
- For a class member, the scope is not local/global, rather we call it private/public (or its variant)
  - There are two more scope in java: package & protected, which we will discuss later
  - If a member is **NOT accessible from outside of the class**/struct, we call the scope of the member as **private**
  - If a member **is accessible from outside function/method of the class**/struct, we call the scope of the member as **public**
  - private & public are the extreme opposite of the spectrum. protected & package scopes lies in between
- In **struct** (c++), **by default** the scope of **ALL** members is **public** (we can change it as per requirement)
- In **C++ class**, **by default** the scope of **ALL** members is **private** (we can change it as per requirement)
- In **Java class**, **by default** the scope of **ALL** members is **package** (we can change it as per requirement)

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Scope summary:

- Scope of Independent variables/objects:
  - global
  - local

```
int main(){
        Student asif;   //scope: local to main
        string str;             //local to main
        return 0;
}
```

- Scope of struct/class member (**field/method**):
  - public
    - if the member is accessible from outside of the struct/class
      - field: can directly read/write
      - method: can be called
  - private
    - if the member is **NOT** accessible from outside of the struct/class

```
class Student{
        int id;                 //private
        String name;    //private
        //assume, Date is an existing class;
        Date dob;               //private
}
```

  - …
  - ….
- in struct, by default everything is public
- in C++ class, by default everything is private
- in Java class, by default everything is package

Q: What is an Object?

A: variable of class/struct

```
int x;
struct/class Node{
        int val;                //member-data of the class    a.k.a field
        Node *next;
};

int main(){
        Node n1, n2;            //n1 & n2 are objects
        cin>>n1.val; n1.next=NULL;
```

```
        //Q: Who set the node? Object or main?
        //A: main. It's not object oriented
}
```

Q: What is Object Orientation?

- Enabling objects to do its own work (like real world objects)
- we can enable objects with functionalities by adding member functions to the class
- member-function a.k.a **method**

```
struct Node{
        private:
        int val;                    //member-data of the class    a.k.a field
        Node *next;
        public:
        void setNode(){
                cin>>val; next=NULL;
        }
};

int main(){
        Node n1, n2;            //n1 & n2 are objects
        //cin>>n1.val; n1.next.NULL;
        //Q: Who set the node? Object or main?
        //A: main. It's not object oriented

        n1.setNode();
        n2.setNode();
}
```

Fundamental characteristics of Object Orientation

- Encapsulation
    - Data hiding/protection
- Inheritance
    - Extension & reuse of existing components
- Polymorphism

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Familiarizing with NetBeans IDE

Coding convention to follow:

- 'abc': Is it a variable name / function name/ class name?

- **identifier** names starts with **small** letters
  - variable, object and function/method names work as identifier
  - identifier identify the memory
    - variable/object name refers to the memory allocated to the variable/object
    - function/method name identifies the starting address of the functions/methods compiled instructions loaded into code-segment of the memory from where the instructions will be fetched for execution
- class name is just a type. type-name itself does not represent any memory, and therefore class name is not an identifier. Thus, as per coding convention, class name starts with Capital letter
- method/function name must sounds like a task/activity, rather that like a noun
- on the other hand, field name represents data/entity, and therefore must sounds like a noun

---

Q: How to display "Hello World"?

Python:

```
print("Hello World")
```

C++:

```
ostream myOut;
//code to connect myOut with a destination to send the output
myOut<<x;
cout<<"Hello World";
=
cout.operator<<("Hello World");        //operator method of ostream class

int id=123;
cout<<"Id="<<id<<endl;
=
cout.operator<<("ID=").operator<<(id).operator<<(endl);
```

Java:

```
PrintStream myOut = new PrintStream();
```

```
//code to connect myOut with a destination to send the output
myOut.println("Welcome");

System.out.println("Hello World");
System.out.println("ID="+id);
```

12 + 13 → 25
"12" + "13" → "1213"
"12" + 13 →"1213"

Q: What is a class 'instance' in the context of OOP?

C++:

```
void doSome(){
        Node n1;
        //when the above line is executed, memory to n1 is already
        //assigned. Therefore, n1 is a class instance (it exists)
}
```

Java:

```
void doSome(){
        Node n1;
        //no memory is allocated to n1 yet.
        //at this point n1 is an empty reference (null), a.k.a class-handle
        //which is capable to refer a future instance to be dynamically created
        //….
        n1 = new Node(); // at this point memory will be allocated from heap
        and now we can say that n1 is a class instance (it exists)
}
```

Function and/or method overloading:

Problem of having so many functions with different names to deal with different data types for same algorithm

```
void sortInts(int a[], int size){....}

void sortFloats(float a[], int size){....}
```

```
int main(){
        int intArr[] = {11,22,33,44,55};
        sortInts(intArr,5);                     //Valid call
        float floatArr[] = {1.1,2.2,3.3,4.4,5.5};
        sortInts(floatArr,5);           //Compilation error
        sortFloats(floatArr,5);         //Valid call
        return 0;
}
```

Solution:

```
void sort(int a[], int size){....}

void sort(float a[], int size){....}

int main(){
        int intArr[] = {11,22,33,44,55};
        sort(intArr,5);                 //Valid call
        float floatArr[] = {1.1,2.2,3.3,4.4,5.5};
        sort(floatArr,5);                       //Valid call
        return 0;
}
```

- To achieve function/method overloading, for each version of the definition, their parameter list MUST be unique:
    - no of parameters can be different
        - void doSome(int);
        - void doSome();
    - type of parameters can be different
        - void doSome(int);
        - void doSome(char);
    - sequence of types can be different
        - void doSome(int,string);
        - void doSome(string,int);
    - combination of all 3 above mentioned options

Class Handle vs Class Instance:

class-instance represents the existence of an object. Therefore, when memory is allocated to an object, it becomes class instance

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

In Java, class-instances are created from heap using dynamic memory allocation (new)

Statement	C++	Java
C+: Student asif, luna("Luna",123,3.5); **Java:** Student asif;	Already asif is an instance, because memory is allocated & constructor is fired	
Java: Student asif, shafiq; asif = new Student("asif",234,3.66); shafiq = new Student();	asif is just an empty reference (null) which will represent a future instance when the memory will be allocated.	//empty handles //asif instantiated //shafiq instantiated

---

Explicit vs Implicit class

Explicit class:

An explicit class is the class which a programmer can explicitly define

Ex:
public class Matrix{...}

Implicit class:

An explicit class is the class which a programmer can NOT explicitly define. These classes are automatically generated by the compiler in machine code

Ex: Java Array classes

int[] intArr;

- intArr is a handle of implicit **one-dimensional-int-array** class
- We can't define this class

```
int[] arr;
public class int[] {
        public int length;
        //…
}
```

```
float[][] table;   //assume table has 3 rows & 4 columns
```

- public class **float[][]** {
  ```
  public int length;
  //…
  }
  ```

```
ComplexNo[] complexArr;
```

- complexArris a handle of implicit **one-dimensional-ComplexNo-array** class
- We can't define this class
  - public class **ComplexNo[]** {
    ```
    public int length;
    //…
    }
    ```

NOTE:

- Whenever we declare an array in java
  - **Compiler generate an implicit array class** for the array that we declared (**in machine-code**)
  - We can't define that array and therefore, there is **no source code available** for the implicit class

Java 1-D Array:

```
class MyUtility{
        void showArray(int a[]){
                for(i=0;i<a.length;i++){
                        cout<<a[i]<<" " ;
                }
        }
}

public class MainClass{
        public static void main(String[] args){
                int arr[]={11,22,33,44,55};
                MyUtility utility = new MyUtility();
                myUtility.showArray(arr);
                return 0;
}
```

implicit array class:

Implicit Array class:

```
public class int[]{
        public int length;
}
```

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Java Array:

In C++:

```
int arr[10];                                    //ok
int arr2[10]={11,22,33,44,55}; //ok
int arr3[]={1,2,3,4};                  //ok
int arr4[];                                     //error
```

Java Statement	Interpretation	Verdict
int x;	x is an integer instance	OK
int[] oneD;	oneD is an array	wrong
	oneD is an instance of **one dimensional implicit integer array** class: int[]	wrong
	oneD is a class-handle of **one dimensional implicit integer array** class: int[]	OK
Now, to instantiate: arrayHandleName = new typeOfElement[noOfElements]; oneD = new int[n];	oneD = new type(no of elements); oneD = new int[](n);	OK
int[][] twoD;	twoD is a class-handle of **two dimensional implicit integer array** class: int[][]	ok
Now, to instanciate: arrayHandleName = new typeOfElement[noOfElements]; int[][] twoD; twoD = new int [rows] [];	twoD = new type(); twoD = new int[][](rows) twoD = new int[][](rows)	Not yet fully instantiated
twoD = new int[rows] []; for(i=0; i<twoD.length;i++) twoD[i] = new int[cols];	twoD = new int[][](rows) loop: twoD[i] = new int[](cols)	Now the twoD is fully instantiated **recommended**
twoD = new int[rows][cols];	Depending on compiler optimization, it may or may not work.	not recommended

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

'this' keyword in java:

- Refers to the **current object (class-instance) in a method or constructor of a class**
- It is an implicit class-handle acts as a reference to current instance
- 'this' is a special reference, which can't be declared and use to refer something as per programmer's choice, because 'this' is an IMPLICIT reference
    - Person asif = new Person();
    - Person emp = asif; **//valid, as emp is an explicitly declared reference**
    - Person this; **//invalid, can't declare 'this' explicitly**
    - this = asif; **//invalid, can't assign asif explicitly to 'this'**

- When we define a class, "this" reference automatically becomes available to the programmer for that class's object
- **"this" will ALWAYS refer to the CLIENT during the LIFETIME of a method call. Once the method returns, 'this' will be NULL**

Method Chaining:

- It is a mechanism, where we can combine multiple method call in a chain
- In method chaining, we can chain multiple calls of same/different methods in one single statement
- Previous call of the chain MUST provide (**by returning**) a **client for the next call** of the chain

Q: Is it mandatory that in a method chain, the client of the chained calls be the same client?

A: No.

Q: Can you give an example?

Example-1:

```
ComplexNo c1, c2, c3, c4;
c1 = new ComplexNo();       c1.setComplexNo();
c2 = new ComplexNo();       c2.setComplexNo();
c3 = new ComplexNo();       c3.setComplexNo();
c4 = c1.add(c2).add(c3);
```

client: c1 client object returned by previous call of the chain: temp

different clients are used for each of the add method call

Example-2:

```
Matrix m1 = new Matrix();
```

```
m1.setMatrix().showMatrix().transposeMatrix().showMatrix();
```

client: c1 client object returned by previous call of the chain: temp

same client is used for all the calls

Example of private method:

```
public class Student{
	//private fields of student
	private void conductSurvey(){....}
	public: Document solveAssignment(Document assign){
		//process assign and create another Document object solution;
		conductSurvey();
		//… code to populate solution
		return solution;
	}
}

public class Faculty{
	//private fields of faculty
	public void giveAssignment(Student s){
		Document assignment1;
		assignment1.setDocument();
		Document solvedAssignToGrade = s.solveAssignment(assignment1);
	}
	public void giveAssignment(Student[] stds, Document a){
		int i;
		for(i=0; i<studs.length; i++){
			Document solvedAssignToGrade = studs[i].solveAssignment(a);
		}
	}
}

public class MainClass{
	public static void main(String[] args){
		Student[] studArr = new Student[35];
		int i;
		for(i=0; i<studArr.length; i++){
			studArr[i].setStudentInfo();
		}
		Faculty skd = new Faculty();
		Document assignment1 = new Document();
```

```
            assignment1.setDocument();

            skd.giveAssignment(studArr, assignment1);
        }
}
```

Why do we need to use a stream instance as a bridge during I/O operation?

int x;
cin>>x;//assume user gives 123 (one hundred & twenty three) as input

- In reality, every key of your keyboard is a character & has an ASCII code
- Therefore the actual input was '1' '2' '3' '\n'

Value	Equivalent binary
123 (perceived as integer)	0000 0000 0000 0000 0000 0000 0111 1011
char '1': ASCII: 49	0011 0001
char '2': ASCII: 50	0011 0010
char '3': ASCII: 51	0011 0011
Carriage Return (CR): 13	0000 1101

- Given input:
  - 0011 0001 0011 0010 0011 0011 0000 1101
- expected value to be stored in memory of x is:
  - 0000 0000 0000 0000 0000 0000 0111 1011
- We need to process the given bytes and convert them (as required) before storing them in actual memory of x
- That's why we need an intermediate buffer to accumulate the input bytes for processing
  - this buffer is an internal field of the stream class which is accessible and processed by the methods of the stream class

---

Constructor:

Constructor: which constructs something (dictionary meaning)
In class/struct: constructor is responsible in constructing object

- int x;
- cout<<x; //garbage
- Student asif; //if no constructor, the fields will be garbage
- Constructor assigns default/initial values to the fields at the time of memory allocation.
- Therefore, constructor does not wait to be called and automatically gets fired while allocation of memory for an objects
    - as a result, constructor don't has any return type, not even void
- The name of the constructor is the name of the class
- Like other methods, constructor can also be overloaded

Developing GUI application using Java FXML

Early use:
- AWT library
- Java Swing library

Current use:
- JavaFX library
    - FXML application use MVC framework

MVC Framework

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

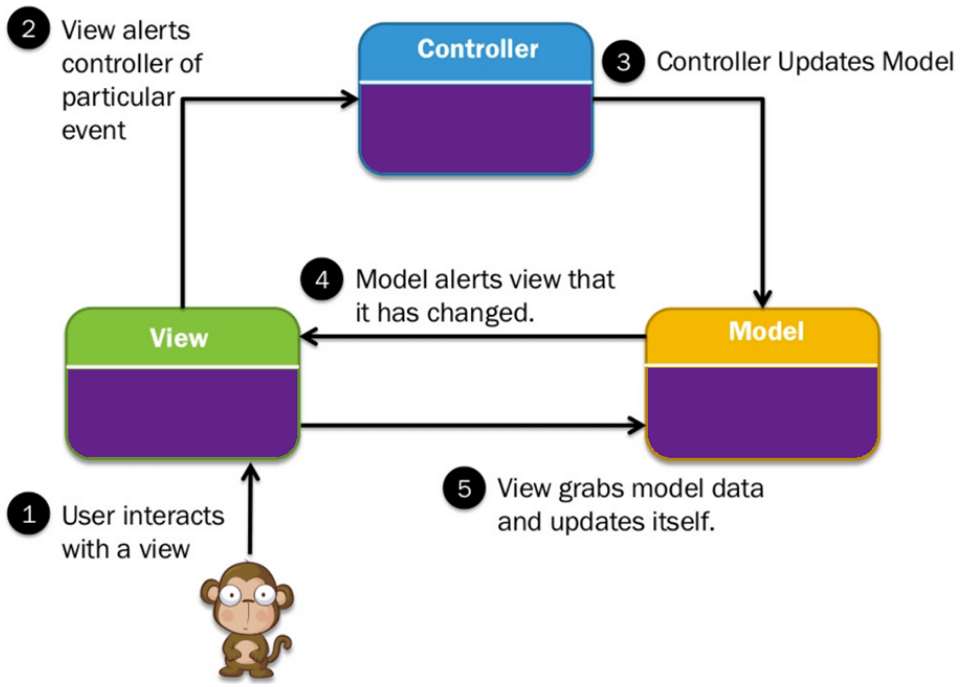


Image reference: https://www.guru99.com/mvc-tutorial.html

Model:

- Classes (.java) representing our application data (Product, Customer, Equipment, Report, etc)

View:

- For web application: Page
- Desktop/Mobile application: Scene
- Elements of a Scene are represented by FXML library classes
- A Scene's code is represented by an FXML file with extension .fxml
- FXML tags are implemented as FX library classes (Button, TextField, Menu, Label, etc…)

Controller:

- Ideally, each fxml file will have a corresponding Controller class (.java)

Q: Why do we need to declare a handle for a class instance?
A: If we want to use the instance later on (in a separate statement after instantiation)
- to access fields / call methods of the class for the instance as client

Q: How many FXML Scenes do you think will be there in your project?
A: Approximately: **70 to 90 Scenes**
- LoginScene of the application : 1
- SingUp Scene : 1
- Home Scene for each User type : 8

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

- Goal specific Scene : 64
- Some supporting scene : 5 to 10

- 70 to 90 .fxml files
- 70 to 90 .java controller class files
- No of Model classes: **25 to 30**
  - 8+1 User classes
  - Let's assume 10 to 15 non-user classes

Project Milestone-1: Customer Requirement Analysis

Deliverable: CRA report

Possible event type:

- UIE - - user input to trigger event
- UID - - user input to be considered as data
- OP – display content (output)
- PC – prerequisite check
- VL – validation check
- VR – verification check
- DP – fetching data from file system and process it to get some calculated outcome (data processing)

Sample workflow for the "**login procedure**" to be used as a part of many other workflows of different goals

	Workflow for login		event type
Workflow:	event-1	User will provide id & password & select login button	UID, UIE
	event-2	id & password will be validated. Validation rule for id is….. & validation rule for password is…..	VL
	event-3	If validation is failed, go back to event-1	…
	event-4	Verify id & pw	…
	event-5	…	…
	event-6	…	
	event-7		

	event-8		
	event-9		
	event-10		

Sample workflow for the goal "**course registration**" for **Student** user of IRAS

User2 name: Student			
Goal-1	**Description of Goal-1:** Student should be able to registrar courses for upcoming semester		event type
Workflow:	event-1 (events of login)	workflow of login process will be executed first	UID, VL, VR: see WF of login
	event-2	if login successful, Student landing/dashboard scene will be loaded	OP
	event-3	select option for course registration	IUE
	event-4	Using student ID, it will verify the date & time clash, payment due & doc pending.	VR, DP
	event-5	If VR is successful, a list of applicable offered courses will be loaded. Each offered course will include: course id, course title, capacity, enrolled, section, time&days if not successful, notification will be given to contact registrar's office	DP, OP
	event-6	While selecting a course one by one, it will check capacity overflow of that course, prerequisite done or not, and time clash with already selected courses	IPD, VR, DP
	event-7	If VR successful,	
	event-8	…	
	event-9		
	event-10		
	…		

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Goal-2	**Description of Goal-2:** xyz				

---

static keyword:

- ### static field
  - when we don't mention static, a field is by default non-static, a.k.a instance-level field. It means, **every instance of the class has its own memory for that field**.

  - Example:

Proposed App Name	**Proposed Class Name**	**Proposed Static Field by the students of CSE213**	**Verdict**
IRAS	Student	uniName	OK
UGC	Student	uniName	Wrong
Jukti			
FoodPanda			
Pathao			
Surokkha			
Some App	SomeClass	someField	
Kasundi	Employee	workplace	OK for single branch, but wrong for multi branch
ProjectMaker	Project	projectCoordinator	Wrong

Dominos	Bill	vatPercentage	OK
MyRobi	Offer	bonusAmount	wrong
NIDPlus	Nationality	birthCountry	depends on context
BKASH	UserInfo	nationality	OK, but implicit default
IRAS	Student	studInfo	wrong
	Identity	nationality	ok for a specific country (unnecessary coz implicitly default), wrong for global identity
	Student		
IRAS/UGC	Student	instituteName	Ok for iras (unnecessary coz implicitly default), wrong for UGC
	Khulna class Division	magura String[] districtNames;	NO
	Contact	telcoName	

- System.out.println();
  - Here out is a static field of System class, whose type is PrintStream. Therefore, the class 'System' is the client for the field
  - println() is a non-static method of the PrintStream class. Therefore the instance 'out' is the client for the method call

• static method

- A static method is a method whose client is the "class" itself, not an "instance". Static method can access static fields ONLY

- Q1: Why do we need an **instance as a client** to call a method?
  - int x=10, y, z=13;
  - Student asif, luna = new Student(); //here asif is not yet instantiated, but luna is already an instance

        - asif = new Student();
        - asif.printCgpa(); //**non-static method**
        - Student.showUniName(); //**static method**
            - if uniName is a static field of Student class
            - also, showUniName() is a static method
    - A: To identify the instance, from where to use the non-static/instance-level fields potentially to be used by the method

    - Q2: Do we need an instance as a client to call a method, if the method is to use ONLY shared/static fields? Such methods are called static method
    - A: No. Since memory is allocated in class-space (instead of instance-space), still the method needs to know the class name. Therefore className will be the client for a static method call

- **NOTE:** Motivation to define a method as a static method:
    - If we want a method to access **static field only**, we define that method as a static method
    - If the **caller don't have a valid instance available at the moment**, but the method needs to be called, then also we can define that method as static
    - Example: public static void main(String[] argos){...}
        - Here JVM is the caller of the main method which does not have any instance of MainClass readily available. Thats why, even if there may not be any static field declared in MainClass, the main method still needs to be static

- **NOTE:** However, a non-static method can access non-static fields as well as static fields

---

Combinations of parameters makes a machine code platform dependent:

- One such parameter: Big vs little endian
    - Big-endian is an order in which the "big end" (most significant value in the sequence) is stored first, at the lowest storage address. Little-endian is an order in which the "little end" (least significant value in the sequence) is stored first.
    - Example:
        int x=1234;
        - //assume bytes 100 to 103 is allocated for x
        - 0000 0000 0000 0000 0000 0100 1101 0010

        - Big-endian:
          byte # 100: 0000 0000
          byte # 101: 0000 0000
          byte # 102: 0000 0100

byte # 103: 1101 0010

- Little-endian:
  byte # 100: 1101 0010
  byte # 101: 0000 0100
  byte # 102: 0000 0000
  byte # 103: 0000 0000

- There are many such parameters which makes a compiled machine code platform dependent

---

Use of multiple DOTs in a statement:

CASE-1:

- It can be found in method chaining (as we have seen earlier)

CASE-2:

- It can be found when a client is not independent, rather its a field of another class

```
public class Date{
        //private fields: day, month, year
}

public class Employee{
        //private fields: int id; String name; Date doj;
}

public class SomeClass{
        public static void main(String[] args){
                Date valentinesDay = new Date(..,...,...);
                Employee asif = new Employee();
                sout(    vanentinesDay.day );
                // client of day is vanentinesDay & vanentinesDay is an independent instance

                sout(    asif.doj.day );
                // use of multiple DOTs, because client of day is doj, & doj's client is asif
        }
}
```

Class Scope:

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Scope	Accessibility	Strictness ranking	C++	Java
private	most restricted: Can be accessed ONLY from inside the class where it is declared In C++: ● struct: default scope of member is public ● class: default scope of member is private In Java: ● class: default scope of member is package	1	yes	yes
protected (special variant of private)	Still restricted to outside world, except the sub/extended class of this class	2	yes	yes
package	package is a folder. Accessible to other classes belongs to same package, but restricted outside of the package In Java: ● class: default scope of member is package	3	NO	yes
public	Most lenient: Can be accessed from anywhere	4	yes	yes

Package Scope:

- C++: Package scope is not relevant in C++
  - test.cpp, includes
    - global main
    - class Student
    - class Faculty
  - After compilation, test.o/test.obj object file will be created which infuse all machine code of main, Student & Faculty in one single file
  - Therefore, there will be no problem to find the machine code of Student class method, if it is called by a Faculty class method
- Java:
  - SomeClass.java, includes
    - class SomeClass (this is the main class containing main method)
    - class Student
    - class Faculty
  - After compilation, 3 machine-code files will be produced for each of the classes, namely (bytecode / class file having extension .class):
    - Someclass.class
    - Student.class

- Faculty.class. files will be created where the machine code of main, Student & Faculty are scattered in three different files
- Therefore, if a Student class method is called by a Faculty class method, it is important that the class files remain together. We ensure that by keeping them in a folder (package). Thus, the package becomes a scope.

Wrapper class:

Each basic data type has a corresponding class equivalent (wrapper class)

Primitive	Wrapper class
int	Integer
float	Float
…	

```
int x;
Integer y = x;   //ok
str = y.toString();
str2 = Integer.toString(123456);
```

Wrapper classes also comes with some useful non-static/static methods to apply on the primitives
For example:

Integer	static String toString(int val){...} String str = Integer.toString(1234); str = “1234”
	static int parseInt(String s){...} int x = Integer.parseInt(“123”); x = 123 //4 bytes integer

```
int x=10, y=20;
cout<< x+y;                          //30
sout(x+y);                           //30
sout("x=" + x);                      //sout("x=10");
```

```
sout( Integer.toString(x)  + Integer.toString(y)  );        //1020
sout( Integer.toString(x)  + y  );                          //30
y = Integer.parseInt("12345");                              //12345
```

Generic class:

The class whose field type is not predefined (generic field). Depending on the instance of the class the field type will be decided at runtime during instantiation.

Example: Non-generic class

```
public class IntStack{
        private int[] vals;
        private int top;
        public IntStack(){
                top=-1;
                vals = new int[10];
        }
        public IntStack(int size){
                top=-1;
                vals = new int[size];
        }
        public void push(int data){
                if(top>=vals.length) sout("Stack Overflow");
                else vals[++top]=data;
        }
        public int pop(){
                if(top!=-1) return vals[top–];
        }
}
```

Note: This IntStack is not a generic class. Therefore we can't use this class to push/pop floats/Strings, etc other than int.

Therefore, it would be wise to make the Stack as generic class, so that it can be used to push/pop any type of data.

Example: Generic class

```
public class Stack<S>{
        private S[] vals;
        //private int size; //in C++, we need size
        private int top;
        public Stack(){
                top=-1;
                vals = new S[10];
```

```
        }
        public Stack(int size){
                top=-1;
                vals = new S[size];
        }
        public void push(S data){
                if(top>=vals.length) sout("Stack Overflow");
                else vals[++top]=data;
        }
        public S pop(){
                if(top!=-1) return vals[top–];
        }
}

psvm(...){
        Stack<int> stk1 = new Stack<int>();          //stk1 will work as int stack
        Stack<int> stk2 = new Stack<int>(20);              //stk2 will work as int stack
        Stack<float> stk3 = new Stack<float>();            //stk3 will work as float stack
        Stack <String>stk4 = new Stack<String>();   //stk4 will work as String stack
}
```

Note: Generic type E/S/T/U/V etc, can be replaced with a real ClassType only. Therefore, for primitives, we have to use wrapper class

```
psvm(...){
        Stack<Integer> stk1  = new Stack<Integer>();
        Stack <Integer> stk2  = new Stack<Integer>(20);
        Stack<Float> stk3  = new Stack<Float>();
        Stack<String> stk4 = new Stack<String>();   //stk4 will work as String stack
}
```

NOTE:

- Typically Most of the (may be ALL) the library collections classes are Generic as well dynamic
- Some commonly used generic-collection classes are:
    - Array
    - ArrayList
    - List
    - Set
    - Map, etc…

Collection class:

The class which dynamically increases its collection capacity. Users need not to declare or worry about the size of the collection (unlike array) as long as memory support is there.

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

"for each" loop:

Advanced "for" loop / "for each" loop: Can be used with a collection ONLY

Syntax:

```
for( typeOfElementOfTheCollection dummyElementName: collectionName ){

}
```

example:

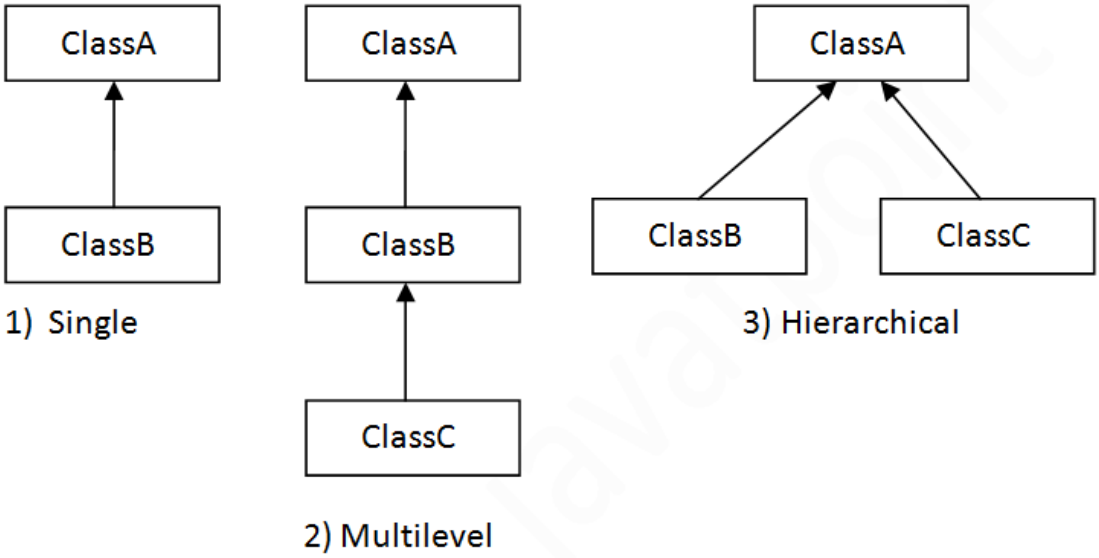
```
int[] vals = new int[10];
for(int x: vals)  sout(x);

ComplexNo[] complexArr = new ComplexNo[20];
for(ComplexNo c: complexArr) c.setComplexNo();
```

Inheritance:

Inheritance is a mechanism in which an object (class instance) of a new class acquires all the properties (fields) and behaviors (methods) from its parent class. It is one of the foundation pillars in object-oriented system design. The idea behind inheritance is that we can create new classes that are built upon existing classes.

- The class which is extended is called: parent / base / super class
- The class which is extending the pre-existing class is called: child / derived / inherited / sub class
- In **java jargon/literature**, they are called as **super** class and **sub** class



- There is another important type of inheritance called **"Multiple Inheritance"**, that we will discuss later
- In real object-oriented design, we often combine the above mentioned inheritance-types to properly represent the system.

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Method overloading vs method overriding:

Method Overloading

- we provide multiple definitions with varying parameter list as part of SAME class
- Aim: To keep ALL overloaded versions ready for execution for the same client. Depending on actual parameter list, one of the overloaded definitions will be executed

Method Overriding

- we redefine inherited method with SAME PARAMETER LIST in subclass
- AIM: If the subclass-author is UNHAPPY with the definition of an inherited method, then the subclass author can define the inherited method in the subclass with the same parameter list.
    - However, this will cause two definitions with the same parameter list available to subclass which will create method overloading conflict. At this point to resolve the conflict, the compiler will DE-ACTIVATE the inherited version & effectively there will be only a redefined version available for execution.
    - this is a win-win situation for subclass author and compiler and this phenomena is called method OVERRIDING

Polymorphism:

Polymorphism means "**many forms**", and it occurs when we have many classes that are related to each other by inheritance.

Like we specified in the previous chapter; Inheritance lets us inherit attributes and methods from another class. Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways. [https://www.w3schools.com/java/java_polymorphism.asp]

Types of Polymorphism:

- **Static polymorphism (static binding of the polymorphic call)**
    - At compile time compiler can decide which version of the method to execute
        - Method overloading (C++ & Java)
        - Operator Overloading (C++)

- **Dynamic/Runtime polymorphism (dynamic binding of the polymorphic call)**
    - It need to wait till runtime to bind a particular definition (among several available versions) for a method call
    - This is because the binding will be decided by the instance of the client of that call

    - Example:
        - public class User{...}
        - public class Student extends User{...}

- public class Faculty extends User{...}

- Let's assume, there is a method in the User class called showInfo() which is overridden in Student and Faculty. Therefore each class has its own version of showInfo().

- **public static void main(...){**
  User u;
  Scanner s = new Scanner(System.in);
  int choice = s.nextInt();
  if(choice==1) u = new User();
  else if(choice==2) u = new Student();
  else if(choice==3) u = new Faculty();

  u.showInfo();
  }

- **Q:** Which version of showInfo() will be executed?
- **A:** Depends on user input

Note: Refer corresponding FXML project involving
- User superclass, and
  - Student & Faculty subclasses done in class

FXML LAB Exercise:
Exploiting polymorphic behavior from the context of an example:

Video Game: Call Of Duty

- Let's assume as a player we collect different weapons such as:
  - Rifle public class Rifle{...}
  - Grenade public class Grenade{...}
  - Pistol public class Pistol{...}
  - Knife public class Knife{...}

- Now, to represent these weapons, we need to maintain 4 different collections (Array, ArrayList, List, Queue, etc) for each type of weapon. For example
  - In main method:
    - Rifle[] rifleArr; OR ArrayList<Rifle> rifleListl;
    - Grenade[] granadeArr; OR ArrayList<Graname> granadeListl;
    - Pistol[] pistolArr; OR ArrayList<Pistol> pistolListl;
    - Knife[] knifeArr; OR ArrayList<Knife> knifeListl;

- Instead, we could use a single array/ArrayList as a collection, if there is inheritance, and use overridden methods to exploit polymorphic behavior of common functionalities.
  - Weapon
    ```
    public class Weapon{
        public void overhaul(){...}
    }
    ```
  - Rifle `public class Rifle extends Weapon{`

```
                @Override
                public void overhaul(){
                        //code to overhaul a Rifle instance
                }
        }
```

- Grenade

```
public class Grenade extends Weapon{
        @Override
        public void overhaul(){
                //code to overhaul a Grenade instance
        }
}
```

- Pistol

```
public class Pistol extends Weapon{
        @Override
        public void overhaul(){
                //code to overhaul a Pistol instance
        }
}
```

- Knife

```
public class Knife extends Weapon{
        @Override
        public void overhaul(){
                //code to overhaul a Knife instance
        }
}
```

- In corresponding controller class, use:
  - ArrayList<Weapon> wList;
  - wList elements can represent any instance of Rifle/Grenade/Pistol/Knife
  - for(Weapon w: wList) w.overhaul();
    - We achieve polymorphic behavior of overhaul() method (if it is overridden in all subclasses)

Q: In our “Call-Of-Duty” example, if we NEVER create instances of Weapon class and the following methods are overridden in ALL subclasses to execute their custom code, then why do we provide definition (body) of overhaul(), showWeapon() & setWeapon() methods in Weapon class?

A: Even though we don’t need the definitions of overhaul(), setWeapon() & showWeapon() in Weapon, still we have to keep them to facilitate overriding in subclasses with the same parameter list & return type (same signature) to achieve dynamic polymorphism.

Q: Then isn’t it the waste of time/code that we define for overhaul(), setWeapon() & showWeapon() in Weapon class?

A: Yes. In that case instead of defining the whole body, we can just declare the prototype of overhaul(), setWeapon() & showWeapon() in Weapon class. That also will ensure FORCED method overriding in subclasses to achieve dynamic polymorphism

---

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Abstract Class & Abstract Method:

- **Abstract method:**
    - Abstract method is nothing but the UN-IMPLEMENTED method
        - method without ANY Implementation/body
        - It is just the prototype of a method
        - Since abstract method don't have any implementation within the super class, then it can't be called using a superclass instance as client (because there is no implementation of the method, and if that method is called, then the program will crash). Therefore, if a class has an abstract method, the class MUST be an abstract class
        - The reverse is not necessarily true. An abstract class can contain non-abstract (REGULAR) methods too. However, the definition of the method will be inherited in subclasses and only subclass instances can be the client of that non-abstract method of the superclass
        - A subclass can override an implemented(non-abstract) inherited method, but it's optional. But it is MANDATORY for the subclass (unless the subclass also is an abstract class) to override all non-implemented(abstract) inherited methods, so that if those methods are called using subclass instance as client, it is guaranteed that there exists some definition to execute.
    - Typically an abstract method is a method of abstract super class, to facilitate polymorphic call of that method using superclass handles
    - If a class contain at least ONE abstract method, the class MUST also be an abstract class
    - Syntax:
        - **public abstract returnType methodName(paraType1 ptName, …);**

-

- **Abstract class:**
    - Class which cannot be instantiated
    - We can only declare its handle & use it to represent subclass instances
    - Syntax:
        - public abstract class ClassName{...}
        - abstract public class ClassName{...}
    - An abstract class may not have ANY abstract method. All of the methods can be implemented (non-abstract) method
    - Consider the following scenario, where it is not possible to achieve polymorphic behavior:
      public class Granade{...}
      public class Pistol{...}
      public class Sword{...}
      ArrayList<Granade> granadeList;
      ArrayList<Pistol> pistolList;
      ArrayList<Sword> swordList;

For example, if we want to maintain a collection of weapons within the context of a video game, where Weapon is a generalization (superclass) but in reality only subclass instances exists within the memory, then we can use Weapon handles to instantiate Granade, Pistol and Sword instances, provided that they are subclass of Weapon. This will allow us to achieve polymorphism (polymorphic behavior of some common methods which are implemented by each of the classes).

```
public class Weapon{...}
public class Granade extends Weapon{...}
public class Pistol extends Weapon{...}
public class Sword extends Weapon{...}
ArrayList<Weapon> weaponList;
weaponList.add(new Granade());
weaponList.add(new Pistol());
weaponList.add(new Sword());
```

- In this example, Weapon qualifies to be an abstract class so that we can define Weapon as an abstract class (instead of non-abstract/regular class):

```
public abstract class Weapon{...}
OR
abstract public class Weapon{...}
```

  - If there is NO abstract method in a class, still the author can declare the class as abstract to prevent instantiation. As a consequence, handle of the class can be declared, but instance of that class CAN'T be created
  - If there is AN abstract method in a class, then it is mandatory for the author to declare the class as an abstract class, too. Now since the class is abstract, then the only role of the class is to act as a superclass to facilitate inheritance. In that case, it is mandatory for the subclasses to override all the inherited abstract methods

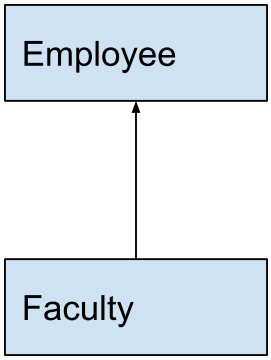
NOTE:

- It is optional for subclass author to override an inherited implemented (non-abstract) method
- But, it is mandatory for subclass author to override an inherited abstract method

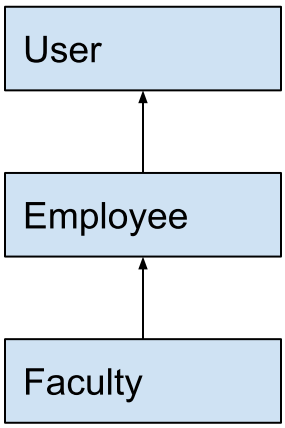
---

Multiple Inheritance

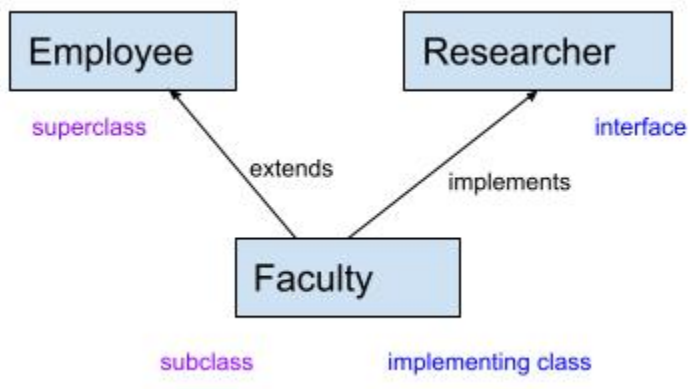
- **Single Inheritance:**
  - A class which has ONLY ONE DIRECT ancestor (super class) and no descendents (subclass) of its own



- **MultiLevel Inheritance:**
  - A class which has ONLY ONE DIRECT ancestor (super class) and ONE or MORE descendents (subclass) of its own



- **Multiple Inheritance:**
  - A class having MULTIPLE DIRECT ancestors (1 super class, rest of them are interfaces)



Interface:

- We can thought of an interface as a cousin of abstract class in the sense that

- An interface can have abstract methods ONLY
- If there are fields, the fields of an interface MUST be:
  - final (field value can't be changed) as well as static (all instance will share same value of the field)
  - Logical existence of such fields are not very common, and therefore majority of the interfaces will not have any field

Q: Why can't we have **multiple classes as ancestors** for a subclass?

A: Allowing multiple classes as superclasses may create runtime problems and cause your application to crash.

- C++ does not care about this and it is the programmer's responsibility to cross check that allowing multiple superclasses will not cause such runtime problems. As a result, C++ allows multiple superclasses for a subclass
- Java is more sophisticated and prevent the application to avoid such problem by NOT ALLOWING MULTIPLE superclasses
  - That's the reason, in java to implement multiple inheritance, only one of the ancestors can be superclass and rest of them has to be interfaces

Example of problem that will be caused from multiple inheritance using multiple superclasses (allowed in C++, not in java):

```
public class Super1{
      public void method11(){...}
      public void method12(){...}
      public void print(){...}
}
public class Super2{
      public void method21(){...}
      public void method22(){...}
      public void print(){...}
}
```

Now **let's assume** that java allows multiple superclass (Actually NOT) and see what happens:

```
public class Sub extends super1, super2{
      public void subMethod(){...}
}
public class MainClass{
      psvm(...){
            Sub obj = new Sub();
            obj.method11();       //OK, as per the assumption
            obj.method12();       //OK, as per the assumption
            obj.method21();       //OK, as per the assumption
            obj.method22();       //OK, as per the assumption
            obj.subMethod();      //OK, as per the assumption

            obj.print();                // Problem, NOT ok
            //2 definitions of print() are inherited from
            //2 superclasses & none of them are overridden version,
            //which will cause definition conflict
      }
}
```

Solution of above problem by using interface in multiple inheritance:

```
public class Super1{
      public void method11(){...}
      public void method12(){...}
      public void print(){...}
}
public interface SomeInterface{
      public abstract void interfaceMethod1();
      public abstract void interfaceMethod2();
      public abstract void print();
}

public class SomeClass extends super1 implements SomeInterface{
      public void subMethod(){...}

      //optional override of implemented inherited method
      @Override
      public void method12(){...}

      //Mandatory override of inherited abstract methods
      @Override
      public void interfaceMethod1(){...}
      @Override
      public void interfaceMethod2(){...}
      @Override

      public void print(){...}    //optional for this case
      //since already an implementation of print is inherited
      //from Super1, now it's a choice for SomeClass whether
      //to override print or not

}
public class MainClass{
      psvm(...){
            SomeClass obj = new SomeClass();
            obj.method11();                  //OK
            obj.method12();                  //OK
            obj.methointerfaceMethod1();     //OK
            obj.methointerfaceMethod1();     //OK
            obj.subMethod();                 //OK
            obj.print();                           //OK
      }
}
```

Example of interface having field

```
public class User{
      //id, name, pw;
```

```
}

public interface Taxable{
      protected static final float highestIncomeTaxRate = 25.0;
      //Not a very good example though, as in future, the maximum
      //tax rate may increase. Therefore, generally no fields are
      //found in almost all interfaces
      public abstract void calculateTax();
}

public class Employee extends User implements Taxable{
      //dept, desig, salary
}
```

Final keyword:

Final	Explanation	Example
field	Once assigned during memory allocation, the value of the field can't be changed	public class Employee{ private final int id; private final Date dob; private String address; private float salary; … }
method	The method which can't be Overridden in subclass (we will discuss method overriding in later class)	TBA
class	The class which can't be extended. As the author of a class, if you don't want to allow extension of your class, then you need to declare that as a final class	public final class Faculty extends Employee{ private int noOfPublication; public void showFacultyInfo(){ showEmployeeInfo(); System.out.println("No of Publication="+noOfPublication); } }

		Faculty class can no longer be extended

Relationships among classes:

- **Composition:**
  - Establishes "**has-a**" relationship
  - Outer class is the **container** (composing) class [ex: Employee]
  - The composing classes's field (which is a handle) is the **contained** (composed) class
- **Aggregation:**
  - Establishes "**has-a**" relationship
  - Outer class is the **container** (aggregating) class [ex: ??]
  - The aggregating classes's field (which is a handle) is the **contained** (aggregated) class
- Difference between composition & aggregation:
  - The main difference between composition and aggregation is **existential**.
    - **Composition:** If container class instance is deleted, then the contained class instance can NO-LONGER EXIST
    - **Aggregation:** If container class instance is deleted, then the contained class instance can STILL EXIST
- **Inheritance:**
  - Establishes "**is-a**" relationship
- **Association:**
  - Establishes neither "has-a" nor "is-a" relationship
  - Here, **one class uses another class** (by calling each other's methods and/or returning values from called methods)
  - They are associated with each other by an activity or task

---

Student example on different relationships among classes:

Project Topic: Library Management System for a public library		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: Employee	Need to rethink
	Contained class name: Librarian	
Composition	Container class name: Book	Need to rethink
	Contained class name: Publisher	
Inheritance	Super class name: Doner	Ok, if the system allows a faculty to donate book to the library
	Sub class name: Faculty	
Association	Class-1 name: Book	Ok
	Class-2 name: Reader	
	Association name: borrows	

Project Topic: Bangladesh Securities & Exchange Commission		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name:BrokerageHouse	Need to rethink
	Contained class name: Client	
Composition	Container class name: Administrator	Need to rethink
	Contained class name: ITDepartment	
Inheritance	Super class name: Branch	Need to rethink

	Sub class name: NewBranch	
Association	Class-1 name: IPOApplication	
	Class-2 name: MarBankManager	
	Association name: evaluates	

Project Topic: Bangladesh Submarine Cable Company Limited		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name:Client	Need to rethink
	Contained class name: ServicePackage	
Composition	Container class name: Employee	Need to rethink
	Contained class name: Task	
Inheritance	Super class name: Report	ok
	Sub class name: ExpenseReport	
Association	Class-1 name: Engineer	ok
	Class-2 name: LandingStationManager	
	Association name: assigns task	

Project Topic: Automobile Workshop		
Type of relationship	**Classes**	**Status**

Aggregation	Container class name:HROfficer	Need to rethink
	Contained class name: EmployeeRecord	
Composition	Container class name: CustomerCareRepresentative	Need to rethink
	Contained class name: Appointment	
Inheritance	Super class name: PurchaseInfo fields: array of product Ids array of product name date of purchase	Need to rethink
	Sub class name: PurchaseType field: payment type	
Association	Class-1 name: Engineer	OK
	Class-2 name: Mechanic	
	Association name: assigns task	

Project Topic: Medical Entrance Exam		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: Admin	Need to rethink
	Contained class name: CandidateApplication	
Composition	Container class name: Admin	Need to rethink
	Contained class name: ExamController	
Inheritance	Super class name: Report	It may be, but salary is more appropriate as a float

	Sub class name: Salary	instead of being a class
Association	Class-1 name: ExamController	OK
	Class-2 name: Candidate	
	Association name: provides QPaper	

Project Topic: Jewelry Chain		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: BranchManager	Need to rethink
	Contained class name: SalesRepresentative	
Composition	Container class name: Customer fields: address, name, contact no, dob, id	ok
	Contained class name: Address	
Inheritance	Super class name: Employee fields: name, id, salary, doj, dob, designation	ok
	Sub class name: SalesRep fields: dailySalesCount	
Association	Class-1 name: Customer	OK
	Class-2 name: ProductRepairRequest	
	Association name: applies/submits	

Project Topic: Law Firm

Type of relationship	Classes	Status
Aggregation	Container class name: Attorney	Need to rethink
	Contained class name: Appointment	
Composition	Container class name: Client	
	Contained class name: ClientData	
Inheritance	Super class name: Lawyer	OK
	Sub class name: Attorney	
Association	Class-1 name: LegalSecretary	OK`
	Class-2 name: CaseFile	
	Association name: submit/updates	

Project Topic: Pragoti Industries Ltd		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: Customer	It is more appropriate as compositio. Because, if we delete a customer, the contact details of that customer will no longer exist
	Contained class name: ContactDetail	
Composition	Container class name: PurchaseOrder	may be, as the app wants to preserve the money receipt for future use in auditing or tax calculation, even if customer cancels the order
	Contained class name: MoneyReceipt	
Inheritance	Super class name: Customer	OK, if CorporateCustomer

	Sub class name: CorporateCustomer	has extra fields and/or methods to be a more specific type customer
Association	Class-1 name: CustomerReport	ok
	Class-2 name: QCManager	
	Association name: generates	

Project Topic: Bangladesh Eye Hospital		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: LabManager	Need to rethink
	Contained class name: DiognosysReport	
	Container class name: Prescription	ok
	Contained class name: String (as medicine name)	
Composition	Container class name: Patient	Need to rethink
	Contained class name: Prescription	
Inheritance	Super class name: Employee	OK
	Sub class name: Nurse	
Association	Class-1 name: ResourceManager	ok
	Class-2 name: InventoryReport	
	Association name: records	

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Project Topic: Model Thana		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: ASP	Need to rethink
	Contained class name: HighProfileInvestigationReport	
Composition	Container class name: Constable	Need to rethink
	Contained class name: FieldNoteOfInvestigation	
Inheritance	Super class name: Designation	Need to rethink
	Sub class name: ASP	
Association	Class-1 name: Inspector	ok
	Class-2 name: Constable	
	Association name: applies for leave to	

Project Topic: Mango Garden		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: Customer	
	Contained class name: OrderHistory	
Composition	Container class name: Product	
	Contained class name: Rating	
Inheritance	Super class name: Mango	Need to rethink
	Sub class name: Amropali	

Association	Class-1 name: Product	OK
	Class-2 name: Customer	
	Association name: buys	

Project Topic: DOHS		
Type of relationship	**Classes**	**Status**
Aggregation	Container class name: LandLord	Need to rethink
	Contained class name: Apartment	
Composition	Container class name:: LandLord	OK
	Contained class name: String	
Inheritance	Super class name: ??	
	Sub class name: ??	
Association	Class-1 name: LandLord	OK
	Class-2 name: Tenant	
	Association name: gives rents to	

Project Topic: Navana 3S		
Type of relationship	Classes	Status
Aggregation	Container class name: Customer	need to rethink
	Contained class name: Complain	
Composition	Container class name: ??	

	Contained class name: ??	
Inheritance	Super class name: Product	OK
	Sub class name: Engine	
Association	Class-1 name: VehiclaInspector	OK
	Class-2 name: Technician	
	Association name: assigns work	

Project Topic: Chain Diagnostic Center		
Type of relationship	Classes	Status
Aggregation	Container class name: Lab	OK
	Contained class name: Chemical	
Composition	Container class name: Lab	Need to rethink
	Contained class name: Machine	
Inheritance	Super class name: Report	OK
	Sub class name: SalesReport	
Association	Class-1 name: RevenueReport	OK
	Class-2 name: SalesReport	
	Association name: uses to calculate revenue	

Project Topic: Metro Rail		
Type of relationship	Classes	Status

Aggregation	Container class name: ShopKeeper	Need to rethink
	Contained class name: Shop	
Composition	Container class name: Shop	
	Contained class name: Location	
Inheritance	Super class name: StationList	
	Sub class name: Station	
Association	Class-1 name: ??	
	Class-2 name: ??	
	Association name: ??	

Project Topic: BSEC		
Type of relationship	Classes	Status
Aggregation	Aggregating class name: Logistic	
	Aggregated class name: Stationary	
Composition	Composing class name: Transaction	
	Composed class name: Invoice	
Inheritance	Super class name: Report	
	Sub class name: AnnualReport	
Association	Class-1 name: ??	
	Class-2 name: ??	
	Association name: ??	

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

UML: Unified Modeling Language

Most commonly, a UML diagram is used **to analyze existing software, model new software, and plan software development and prioritization**. Simply put, if you need a way to visualize and plan your software development process, a UML diagram is incredibly helpful.

source:
https://www.google.com/search?q=why+we+need+uml+diagrams&rlz=1C1PNBB_enBD971BD971&oq=why+we+need+UML+diagrams&aqs=chrome.0.0i512j0i390l3.5827j0j7&sourceid=chrome&ie=UTF-8

UML Diagrams – Everything You Need to Know to Improve Team Collaboration

- What is a UML Diagram?
- What is a UML Diagram used for?
- Types of UML Diagrams

Source:
https://www.bluescape.com/blog/uml-diagrams-everything-you-need-to-know-to-improve-team-collaboration

---

How to create UML Class Diagram from **Customer Requirement Analysis report** (functional specification)?

- First we will explore an online tool to create the UML class diagram
  - lucidchart
- Initially, we will identify the classes by analyzing workflows of CRA report
- Understand how to normalize a class (breaking down/ restructure an initially formed class to keep the redundant fields at bare minimum across different classes),
  - and then we will normalize our initially identified classes
  - Consider the post normalized classes to create class diagram
- Use IRAS example and use a hypothetical CRA report of IRAS to produce UML class diagram for IRAS
  - due to time constraint, we will work on partial requirement to produce partial class diagram

---

Representing Various Components in UML Class Diagram:

Type: Class / Abstract Class / Interface:

- Represented by a box
- Has 3 compartments (2 for interface):
  - Top compartment will have Class/Interface name
  - Middle compartment will have fields
  - Bottom compartment will have methods

Access specifiers (visibility):

private	**-**
protected	**#**
package	**~ (tilde)**
public	**+**

Modifiers:

Description	**Notion**	**Example**
Static	underline	#/- activeMemberCount: int
Read only (final)	{readOnly} or {final}	#/- dateOfJoining: Date {readOnly}
unique	{unique}	email: String {unique} ID: int {unique, final}
collection	collection [ ]	visitors: String[*/n] {unique}

Inheritance:

- Solid-line arrow from subclass to superclass (arrow is indicated with unfilled triangle)
- Dotted-line arrow from implementing class to interface (arrow is indicated with unfilled triangle)

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

- Abstract class name : italic
- Abstract method name : italic
- Non-abstract (concrete) class : normal font
- Non-abstract (concrete) method : normal font
- Interface: : <<Interface>> followed by interfaceName

Composition/Aggregation:

```
class Booking{
        Member m;   //aggregation
        Payment p;  //composition
}
```

Composition : Solid line with filled diamond at one end, diamond towards composing (container) class

Aggregation : Solid line with unfilled diamond at one end, , diamond towards aggregating (container) class

Multiplicity:

- Can be applied on: composition, aggregation & association
- Multiplicity can be:
  - exactly 1 (mandatory 1)
    - denoted by: **1** OR **1..1**
      - place 1 on applicable side (for association, if not unidirectional)
      - place 1 on composed/aggregated class side (for composition/aggregation)
  - 0 or 1 (optional 1)
    - denoted by: 0..1
  - 0 or * (optional many)
    - denoted by: 0..* OR 0..n (if n is the upper bound)
  - 1 or * (at least 1, mandatory many)
    - denoted by: 1..* OR * OR 1..n (if n is the upper bound)

Association:

Customer places Order

Used straight line (no arrow needed for bi-directional, and **Non-triangular-open-arrow** for uni-directional)

Example of Association:

Notification [0..* ?? isSentTo] 1..* Member

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Booking	[canMake ?? doneBy] [0..* 1]	Member
Customer	[orderedBy places canOrder] [1 0..*]	Order
Member	[refers] [1 0..*]	Member

Examples of multiplicity (for association):

Class-1	multiplicity - associationName - multiplicity	Class-2	Direction
Employee	[1 **uses** 0..1]	OfficeComputer	uni-directional
Faculty	[1..16 uses 1..5]	ClassroomPC	uni-directional
Buyer	[0..* views 0..*]	Property	uni-directional
VideoEditor	[0..* usesCPUforRendering 1..*]	CPUCore	uni-directional
GymMember	[1 refers 0..*]	GymMember	reflexive

Examples of multiplicity (for composition/aggregation):

CONTAINER class: for composition/ aggregation)	multiplicity -- multiplicity	CONTAINED class: for composition/ aggregation)
Car	[1 1]	Engine
JetAircraft	[1 2..n]	Engine
Researcher	[1..* 0..*]	Publication
MotherBoard	[1 1..*]	RamSlot
Library	[1 1..*]	Resource
MobilePhone	[1 0..1]	ESim

File I/O in Java:

Input/Ouput (I/O):
==========

- **Input:** writing content to the memory (input for memory)
- **Output:** reading content from the memory (output from memory / literal)

- **Console I/O:**
  - Input (write): We take input from console input device (by default: keyboard) & store in memory (variable/instance)
    Ex:
    cin>>x; [C++]
    Scanner s = new Scanner(System.**in**); x = s.nextInt(); [Java]

  - Output (read): We read content from memory (variable/instance) & send output to console output device (by default screen)
    Ex:
    cout<<x; [C++]
    System.**out**.print(x); [Java]
- **File I/O:**
  - We take input from a file (no default, we have to specify the source file) & store in memory (variable/instance)
  - We read content from memory (variable/instance) & send output to a file (no default, we have to specify the destination file)

For ANY KIND of Input, we need to use one of the suitable INPUT-STREAM CLASS INSTANCE as a bridge between the **source of input** and **memory**

For ANY KIND of Output, we need to use one of the suitable OUTPUT-STREAM CLASS INSTANCE as a bridge between the destination of output and memory

I/O in C++:
==========

Type of I/O	**Stream class name**	**Purpose**
Console input	istream	writing to memory, from KB
Console output	ostream	reading from memory, to screen
File input	**ifstream**	writing to memory, from a specified file
File output	**ofstream**	reading from memory, to a specified file
File Input & Output	**fstream**	reading and writing- using same stream from/to a file

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Console I/O in Java:
==========

Type of I/O	Stream class name	Purpose
Console output	PrintStream: (System.out)	reading from memory, to screen

Ex:
cout<<x; [C++]
System.out.print(x); [Java]
- we are using stream class's method both in C++ & Java

Console input	InputStream: Scanner(System.in)	writing to memory, from KB

Ex:
cin>>x; [C++]
Scanner s = new Scanner(System.in); x = s.nextInt(); [Java]
- we are using stream class's method in C++, but NOT in Java for convenience

- instead of using read methods of InputStream class, [System.in.read();]
  We use Scanner class's methods.
- While instantiating Scanner instance, we provide InputStream instance (System.in) to the constructor of Scanner as the source of scanning for convenience.

File Handling in Java (File I/O):

- Text file:
- Stores character, so that any text editor can open & read the content as characters
  Ex: If your ID is 1830289, in text file it will be stored as
  "1830289", content size will be 7 bytes for 7 characters

- Binary file:
- Stores raw data
  Ex: If your ID is 1830289, in binary file actual bytes of the value will be stored.
  therefore, content size will be 4 bytes as the input is an integer

Difference between text & binary file:

- Storage device stores: binary equivalent (irrespective of file type)
  - SSD/FlashDrive/RAM: presence/absence of electric charges
  - Magnetic drive: magnetized/demagnetized cell
  - optical drive: high/low intensity laser pits

- Text file:
  - Every bytes of a text file is considered as the ascii code of a character

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

- When we open a text file using a text editor (notepad, subline, word,...)
  it shows corresponding character for the bytes stored in that file
- Therefore, we use text file to store textual data only

- Binary file:
  - Store raw data (actual bytes of the content):
    - image file (collection of pixel bytes)
    - Audio file
    - video file
    - writing/reading instances to/from file
    - database

Q: What type of file you will use for your project
A:
- Mostly we will use binary files (for application data)
- A few text files (readme/help/FAQ/installationGuide, etc)

Why we will use binary file for our project:

Suppose we have an int x = 123
binary of 123 (4 bytes) is: 00000000 00000000 00000000 01111011
these 4 bytes need to be converted to bytes (ascii code) of character '1', '2' & '3'

binary of "123": 00110001 00110010 00110011
ascii:49 50 51

- While writing: Integer.toString(..); //additional task
  Conversion: 00000000 00000000 00000000 01111011 --> 00110001 00110010 00110011

- While reading : Integer.parseInt(..);//additional task
  Conversion: 00110001 00110010 00110011 --> 00000000 00000000 00000000 01111011

- Also, keeping application data in text file will allow anyone to view the content using any text editor. But the content of a binary file can't be retrieved unless the user knows the **metadata** of the binary file.

- We can also have some text file in our project such as "readme.txt", "faq.txt", "about.txt"

Streams that we can use to perform file I/O in java:

- Byte stream [Both for text & binary file]
- read/write raw bytes from/to file
Classes:
Reading: **FileInputStream**
Writing: **FileOutputStream**

- Character stream [For text file, recommended]
- read/write characters from/to file

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Classes:
Reading: **FileReader** or **Scanner** for convenience (File will be source of Scanner)
Writing: **FileWriter**

- Data stream [For binary file]
- read/write different data types as it is (their actual bytes) from/to file
Classes:
Reading: **DataInputStream**
Writing: **DataOutputStream**

As an intermediate layer to increase read/write efficiency, we can use Buffer in between Data & Byte streams
Classes: BufferedInputStream & BufferedOutputStream

- Object stream [For binary file]
- read/write deserialized/serialized instances from/to file
Classes:
Reading: **ObjectInputStream**
Writing: **ObjectOutputStream**

Text File:
- We can use Byte stream (if it is 1-byte ASCII character) as well as Character stream
- However, character stream is recommended for text file

Binary file:
- Data & Object stream will be used for your project. However, these Data & Object stream will internally use Byte stream
- To generate pdf/jpg --> Byte stream
- int/float/String & mix of them --> Data stream
- Only class instances --> Object stream

NOTE: To perform file IO, we need to deploy Exception Handling mechanism

Exception Handling:

```
int main(){
    int a,b;
    float ,c;
    cin>>a>>b;
    c = (float) a / b;          //critical statement, division by zero, exception
    cout<<a<<b<<c;
    return 0;
}

//we can avoid this / by 0 situation by optimizing our source code,
//Without deploying an Exception handling mechanism.
int main(){
    int a,b,c;          cin>>a;
    do{
        cin>>b;
```

```
            }while(b==0);
            if(b!=0) c = a/b;          //critical statement
            cout<<a<<b<<c;
            return 0;
        }
```

NOTE:

- So far, the programs that we have written had some critical statements, and by analyzing the source code, compiler knows potential exception from those critical statements can be avoided by **optimizing the source code** [UNCHECKED Exception].

- Therefore, the compiler **never forced us to deploy EH-mechanism** for potential UNCHECKED exceptions.

- **BUT,** there can be some critical statements which might fail during runtime due to some external factors, which is beyond the control of the programmer, and therefore it may not be possible to avoid that situation by optimizing our source code. [CHECKED Exception]

- For these critical statements having potential for **CHECKED exception**,
  compiler forces us to deploy EH-mechanism:
  - using **try-catch-finally** block

- File I/O related exceptions are CHECKED exception,

- **File handling is one such case, where exception handling is enforced.**

What is Exception???

In C++: It can be any data type
In Java: It is an instance of ONE-OF-THE Exception class family (inheritance)

```
int cppArr[5] = {1,2,3,4,5};
        cout<<cppArr[2];        // 3
        cout<<cppArr[5];        // garbage

int[] javaArr = {11,22,33,44,55};
        sout(javaArr[2]);       // 33
        sout(javaArr[5]);       // jvm wll throw ArrayIndexOutOfBoundsException,
                                // if the thrown exception is not handled, program will crash


public class SomeClass{
        public void someMethod(){
                int a,b; float c;
                Scanner s = new Scanner(System.in);
                a = s.nextInt(); b = s.nextInt();

                c = (float) a / (float) b;
                // if b is zero, and code is not optimized, so, / by 0 will occur
                // at this point an ArithmeticException instance will be thrown implicitly by the JVM
```

```
                // and since there is no handling block (catch block), the program will crash immediately

                sout(...);          //this line and lines below will never be executed..
                ......
        }
}



public class SomeClass{
        public void someMethod(){
           try{
                Scanner s = new Scanner(System.in);
                a = s.nextInt(); b = s.nextInt();

                c = a/b;
                // if b is zero, and code is not optimized, so, / by 0 will occur
                //at this point an ArithmeticException instance will be thrown implicitly
                //and since there is no catch block, the program will crash immediately

                sout(...);          //this and lines below will never be executed..
                ......
           }
           catch(ArithmeticException e){
                //exception handling code
                //e.someMethodOfArithmeticException();
                //e.someGetter(); //to get exception related info to show to the user
                //your own handling code...
           }
           catch(RuntimeException e){
                //your own handling code...
           }
        }
}
```

- You can also have your own exception:

```
        public class MyException extends AnyOfTheExceptionClassFromExceptionHierarchy {

        }
```

Five keywords that we need to understand for exception handling:

try:

- represents the block where the critical statements are kept
  (we try to execute critical ststemets here)
- can have nested try blocks
- while trying to execute the code inside try block, if an exceptional context arise
  a suitable Exception type instance is thrown by the JVM. Therefore,
  there must be atleast one (or more) corresponding handling block (catch)
  associated with the try block.

- the thrown instance contains runtime contextual data of the exceptional situation as fields, which can be used in catck block for amicable solution
- Once an exception is thrown at a critical ststement, control will leave the try block and as a result, rest of the statements of the try block will NEVER be ececuted

catch:

- exception handling block (handling code are kept here)
- When multiple catch blocks are associated with a try block, we keep the:
  MOST specific catch block (subclass) at the top, and
  MOST generic (superclass) catch block at the bottom.

finally:

- To be discussed in later class

throws:

- To be discussed in later class

throw:

- To be discussed in later class

Inappropriate order of the catch blocks:

```
try{
        //critical statements
}
catch(Exception e1){...; e1.someMethod();....;}
catch(RuntimeException e2){...; e2.someMethod();....;}
catchArrayIndexOutOfBoundsException e3){...; e3.someMethod();....;}
catch(ArithmeticException e4){...; e4.someMethod();....;}
catch(IOException e5){...; e5.someMethod();....;}
catch(EOFException e6){...; e6.someMethod();....;}
```

Right order of the catch blocks:

```
try{
        //critical statements
}
catch(ArrayIndexOutOfBoundsException e3){...; e3.someMethod();....;}
catch(ArithmeticException e4){...; e4.someMethod();....;}
catch(RuntimeException e2){...; e2.someMethod();....;}
catch(EOFException e6){...; e6.someMethod();....;}
catch(IOException e5){...; e5.someMethod();....;}
catch(Exception e1){...; e1.someMethod();....;}
```

Milestone-2: System Design:

- It includes many things, but we will restrict ourselves in producing following deliverables:
  - UML class diagram : PDF
  - Description of File structure (representing database) : Docx
- To produce the above, we need a comprehensive CRA-report in place (Milestone-1)

Steps to built class diagram:

- Understand the growth volume of the class-instances and how to normalize the high-growth volume classes
  - (We will try to understand this by our own terms "Master" and "Non_Mster/Transaction" classes)
  - Avoid redundant fields to restrict volume
  - Master class typically will not have any foreign field (field of another class)

Example of Master Class:

```
public class Course{ //represents a course of IUB curriculum
        private String courseID, title, courseType;
        private int noIOfCredits;
        private String[] prerequisiteCourseIds;
        Document courseOutline;
}
Q: Is there any foreign (redundant) field?
A: No                                   //OK
Q: Is there any unique field?
A: Yes, courseID                        //OK
Q: Is it a high-growth volume class?
A: No
Q: Is there any redundant field?
A: No
```

Example of Non-Master Class:

```
public class OfferedCourse{ //represents a offered course offered for registration
        private String courseID*, title*, roomNo, days, time, facultyName*, semester;
        private int noOfCredits*, sectionNo, capacity, enrolled=0, facultyID*;
        private String[] prerequisiteCourseIds*;
        Document courseOutline*;
}       //*: foreign fields
Q: Is there any foreign field?
A: Yes                                  //OK, required
Q: Is there any unique field?
A: No                                   //Doesn't matter. May or may not exist
Q: Is it a high-growth volume class?
A: Relatively yes
Q: Is there any redundant field?
A: Yes
```

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Master vs Non-Master class:

Master class:

- Class which has a unique field (value of the field is unique for each instances) to identify an instance.
- The growth of instance volume (database size) is under control / insignificant
- Also, there is no redundant field

Non-Master (Transaction) class:

- Class which has NO mandatory unique field (no such field, whose value is unique for each instances) to identify an instance.
- Also the growth of instance volume (database size) is relatively high
- Typically, it will have redundant fields

Example: Dutch Bangla Bank Limited:

#: No of daily new Account (class) is opened for the bank:

- No of branch: 214
- No of Fast Track booth: 1268
- No of ATM: 4930
- No of Agent banking: 63
- Assume, no of new accounts opened in a branch: 100
- Assume, no of new accounts opened in DBBL Fast Track booth: 50
- Assume, no of new accounts opened through DBBL Agent banking: 30

- No of new accounts: 214*100 + 1268*50 + 63*30 = 67,430
- In DBBL data center, 67,430 new records/day (Account class instances) are added to the database

#: No of daily bank-transactions occurred in DBBL:

- No of cash deposit in a branch: 500	* 214	= 107000
- No of check deposit in a branch: 700	* 214	= 149800
- Deposit cash using CDM in fast track: 100	* 1268	= 126800
- Deposit check using CDM in fast track: 200	* 1268	= 263600
- Online fund transfer via internet banking: 50000		= 50000
- Online fund transfer via rocket (mobile banking): 10000		= 10000
- Withdraw cash from each ATM: 300	* 4930	= 1479000
- Encashing check from each branch: 700	* 214	= 149800
- Withdrawal as a result of online fund transfer:		= 50000
Total bank-transactions of DBBL per day:		= 2341000

- Growth of volume: 67,430 vs 2341,000

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

public class Account{ //master class
fields: **accountNo**, accountName, typeOfAccount, balance,
address, email, contactNo, NID, ...
}
- has **unique** field
- no redundant field (as foreign field of another class)
- volume growth under control
- It is a Master class
- No redundant field

public class BankTransaction{
fields: locationId, amount, typeOfTransaction, date, time, processedBy,
accountNo*, accountName**, contactNo**
} //* **redundant field**, * **unnecessary redundancy**
- No unique field
- volume growth is quite high
- has **redundant** field
- It is a Transaction class

- Since volume growth is quite high, we need to normalize the class by eliminating redundant fields
- Also we can add an unique identifier (new unique field) to the class as follows

public class BankTransaction{ //Normalized class
fields: transactionId,
accountNo, locationId, amount, typeOfTransaction, date, processedBy
}

Analyzing CRA report:

Identify Model classes/interfaces representing application data:

- **Identify USER classes**
  - All the users are classes
  - All classes will have constructors, getter, setter, toString as default
  - All the **goals** of an user are the **methods** of that user class
  - All

- **Identify NON-USER classes**
  - It's ***easier*** to detect non-user MASTER classes from the workflow
  - To detect non-user non-master (potentially unnormalized) classes, we need to understand that the fields of these classes represent datasets produced as the byproduct of an activity. Therefore we need to carefully analyze the workflow to detect the non-master classes.

- **Identify interfaces**

- **Finalize decision about the user defined type**
  - Decide whether the type will be a:
    abstract-class / non-abstract class / interface

- Decide whether the class is final or not

- **Finalize decision about the fields**
  - Decide whether the field is static or non-static
  - Decide whether the field is final or not

- **Finalize decision about the methods**
  - Decide whether the method is static or non-static
  - Decide whether the method is abstract or not
  - Decide whether the method is final or not
  - ## Does this method use user input from view (controller classes' private fxids) or not?
    : if yes, the user input should come as parameter from handler method
  - ## Does this method return data/status back to the UI-scene or not?
    : this will guide to decide the return type of the method

- **Identify NON-USER classes**

- **Identify Controller classes representing UI-scene:**
  - We will talk more on this when we start building FXML application (next class)

- After detection of classes, we need to establish relationships among those classes:
- inheritance
- aggregation
- composition
- association and,
- multiplicity (0:1, 1:1, 1:M, M:1, M:M, including mandatory/optional, and for M: finite/infinite)

- Then draw the UML class diagram for the above using one of the online tools. We will use lucidchart

---

Writing workflow for a sample goal (IRAS):

User: Student
- **Goal-1**: Register courses for upcoming semester
  - Workflow:

e1: check login credential (give detail....)
- ..
- …

e2: if login successful, load Student-specific homeScene, else go-back to login/forgotPW

e3: after selecting "register course" option, it will check the following:
- document pending
- date & time-slot clash
- probation-status.

If verified positive, a new scene will appear which will be pre-loaded with ALL applicable offered-course (info of offered courses such as: courseID, title, roomNo, days, time, facultyID, facultyName, semester, noIOfCredits, sectionNo, capacity, enrolled, prerequisiteCourseIds...) for the student. **Else**, it will prompt the student to contact the registrar office.

e4: Student will select ONE-Course at a time and it will check:
- capacity, time-clash-with-already-selected-courses

e5: proceed to register, and it will check:
- final Check of capacity for all selected courses
- no of credit taken is within the range [9 to 18]
- for "Course/Lab" paired courses whether one is left out or not
- if any fail course is omitted

e6: If the above checks are positive, it will create RegisteredCourse details, commit to the database and generate the bill.
Bill content: .....

Drawing UML class-diagram (partial) using CRA-Report (IRAS):

public class Student: [has unique field, Master class]

public class Faculty: [has unique field, Master class]
- **facultyId**, facultyName, desig, dept, salary, doj, dob,....

public class Course: [has unique field, Master class]
- **courseId**, courseTitle, noOfCredits, preRequisiteList,...

```
public class CourseTimeStamp{
    year, semester, day, time;
}
```

public class OfferedCourse: [Unnormalized Transaction class]
- *courseId*, ~~*courseTitle*, *noOfCredits*~~, sec, year, semester, day, time, *facultyId*, ~~*facultyName*~~, roomNo

public class OfferedCourse: [Normalizing non-master class, eliminating redundant fields]
- courseId, sec, year, semester, day, time, facultyId, roomNo

public class RegisteredCourse: [fields StudentID, courseID, timeStamp, section]

public class RegistrationBill:
- billNo, totalAmount
- public void generateBillPdf(){
    use "registeredcourse.bin" file rom database
  }

Example:

RegisterCourseScene.fxml

```
class RegisterCourseSceneController .....{

        ArrayList<CourseTimeStamp> selectedCourseTimeStamp;
        ArrayList<Integer> selectedCourseIDs;

        proceedToRegisterCourseButtonOnClick(...){
                //commit to database
                write to file: "1234567-registeredcourse.bin"
                int <student id>
                int<1th-coutseId>, selectedCourseTimeStamp-instance
                int<2th-coutseId>, selectedCourseTimeStamp-instance
                int<3th-coutseId>, selectedCourseTimeStamp-instance
                int<4th-coutseId>, selectedCourseTimeStamp-instance
                .....
                int<nth-coutseId>, selectedCourseTimeStamp-instance
                //arraylist end here
                //end of file writing
        }

}
```

Student.java

```
public class Student exteds.... implements...,...,...{
        //fields

        /*
        public void setStudentInfo(){
                //id = user input from console: OK
                //name = user input from TextField of controller class: NOT OK,

                //TextField fxid is private in controller class, NOT accessiable to method of Student class
        }
        */

        public boolean setStudentInfo(int id, name String,...){
                this.id = id        //OK
                //TextField valuue of controller class is/are passed as parameter
        }
}
```

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Lab Exercise:

IRAS users: Student, Faculty, LibraryOfficer, AccountsOfficer, RegistrationOfficer, AdmissionOfficer, FinalcialAidOfficer, HOD, Dean, OfficeManager, VCOfficeExecutive, VC, ProVC, Treasurer, BoTMember, CITSAdmin, ...

For the Student dashboard of IRAS, we will try to identift the elements needed to draw UML class diagram

- Student is a user of IRAS

Goal-1: Check attendance of a course
Goal-2: Update profile information
Goal-3: Declaring major/minor
Goal-4: Pay bill
Goal-5: Register course for upcoming semester
Goal-6: Add/Drop course for current semester
Goal-7: Withdraw course for current semester
Goal-8: Generate transcript
Goal-9: Evaluate Faculty
Goal-10: Apply for financial aid
Goal-11: .....

Classes:

User classes:

User.java:

```
public abstract class User{


}
public class Student extends User{ //TODO: your code }
public class Faculty extends User{ //TODO: your code }
public class HOD extends User{ //TODO: your code }
public class RegiatrationOfficer extends User{ //TODO: your code }
public class FinancialAidOfficer extends User{ //TODO: your code }
public class AccountsOfficer extends User{ //TODO: your code }
```

Non-User classes:

```
public class Course{ //TODO: your code }
public class OfferedCourse{ //TODO: your code }
public class RegiateredCourse{ //TODO: your code }
public class AttepmtedCourse{ //TODO: your code }
public class Document{ //TODO: your code }
public class Bill extends Document{ //TODO: your code }
public class Transcript extends Document{ //TODO: your code }
public class Assignment extends Document{ //TODO: your code }
public class Notice extends Document{ //TODO: your code }
Public class Scholarship{ //TODO: your code }
```

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

Guideline to solve the exercise:

```
public abstract class User{
        protected fields: userid, name, dob, bg, gender, nid, password, email, address


        public abstract boolean setUserInfo(.....);                    // [;/{}]
        final public boolean login(....){...}          //final method


        //public abstract void applyForLeave();
        //It will force BoTMember to implement applyForLeave, which is irrelevant for them
        //Therefore, we will keep the abstract method in an interface
}
```

LeaveApplication.java

```
public interface LeaveApplication{
        public void applyForLeave();
}
```

Student.java:

```
public class Student extends User{
        private int studId; private String major, minor;
        private ArrayList<AttemptedCourse> grades;   //for option-1 of AttemptedCourse class
        public ?? checkAttendance(...)??
        public ?? updateProfile(...)??
        public ?? payBill(...)??
        public ?? evaluateFaculty(...)??

}


public class Student extends User implements LeaveApplication, CourseRegistration{          }

public class Faculty extends User implements LeaveApplication, CourseRegistration{}

public class FinalcialAidOfficer extends User implements LeaveApplication{   }

public class RegistrationOfficer extends User implements LeaveApplication, CourseRegistration{      }


public class BoTMember extends User{          }

public interface I1{       }
Q: Can we do the following extension?
        ..... extends I1{ }
A: Yes/No
        public class C2 extends I1{      }: NOT OK
        public interface I2 extends I1{  }: OK
```

Independent University, Bangladesh
Class scratch pads for Object-Oriented Programming-I course (CSE213)

```
public class Bill{          }

public class Course{    }

public class OfferedCourse{     }

//option-1 (composition), where Student maintains a collection of AttemptedCourse instances
public class AttemptedCourse{
        courseId, semester, sec, letterGrade
}

//option-2 (association), where ALL grades instances will be dumped in a single table of the database
public class AttemptedCourse {
        studId, courseId, semester, sec, letterGrade
}

public class Scholarship{         }
```