2021-11-02 -- Scratchpad of CSE213 (Sec-1)
==========

Difference between method overloading and method overriding:

- In overloading, All versions of the method is available for execution for SAME client,
  but in overriding inherited version can never be called for subclass client

- For overloading, ALL versions must have unique parameter list, but in overriding both
  inherited and own version has same parameter list (hence to avoid ambiguity, deactivate
  inherited one)

- Overloaded versions may be written in same class, OR
  some of them can be written in superclass and some of them can be written in subclass.
  BUT, overridden versions will be written: one in superclass, and another in subclass

==========

Relationships among classes:
---

a) Inheritance
    - Extension of preexisting superclass

b) Aggregation
    - A class contains a handle of another preexisting class, as its field
    - Establishes "has-a" relationship
    - Aggregation implies a relationship where the aggregated (which is a Preexisting class's
      handle as a field of new class being defined) class instance can exist independently
      of the aggregating (the new class) class instance.

        Example-1:

        - Toilet is a preexisting class
            public class Toilet {....}

        - Auditorium is the new class being defined
            public class Auditorium{
                private Toilet[] toiletArr;
            }

        - Here Auditorium is aggregating class, and Toilet is aggregated class,
          and Toilet instances exists outside/independent of Auditorium in IUB

```
                Example-2: Totota Corporation
                ----------------------------
                 - Engine is a preexisting class
                        public class Engine {....}

                 - Car is the new class being defined
                        public class Car{
                                private Engine eng;
                        }

                - Here Car is aggregating class, and Engine is
aggregated class,
                  and Engine instances exists outside of Car instance
as SPARE paers in Toyota



c) Composition
        - A class contains a handle of another preexisting class, as
its field
        - Establishes "has-a" relationship
        - Composition implies a relationship where the composed (which
is a Preexisting class's
           handle as a field of new class being defined) class
instance can NOT exist
           independently of the composing (the new class) class
instance.

                Example-1:
                ----------
                 - Engine is a preexisting class
                        public class Engine {....}

                 - Car is the new class being defined
                        public class Car{
                                private Engine eng;
                        }

                - Here Car is composing class, and Engine is composed
class,
                  and Engine instances Cannot exists outside of Car
instance in IUB

d) Association
        - When One class's method interact with another class's method
(using one another),
          but there is no "is-a" / "has-a" relationship, then we call
it association


================================================================
=================

Polimorphism:
-------------
        - Poly: many
```

- Morphing: Smooth transition of image to different form

- Polymorphism in Java is the ability of an object to take many forms.
To simply put, polymorphism in java allows us to perform the same action in many
different ways. It is a feature of the object-oriented programming, which allows a
single task to be performed in different ways (depending on runtime context).

- Inheritance is involved in acheiving polymorphism (as superclass handle can be used
to represent any subclass instances of the inheritance hierarchy)

Example:

Video Game: pubji, Valorent

- Assume, a warfare game has different types of weapons as resources for the player

```
public class Granade{...}
public class Pistol{...}
public class Sword{...}
```

- To preserve the collected weapons, there need to be 3 DIFFERNT ARRAYs
(more dynamic collection alternate of Array) for Granade, Pistol & Sword

```
Granade[] granadeArr;
Pistol[] pistolArr;
Sword[] swordArr;

granadeArr[i] = new Granade();
PistolArr[i] = new Pistol();
swordArr[i] = new Sword();
```

- But if we want to maintain only ONE ARRAY instead of three, then we can
introduce a superclass called Weapon, and make Granade, Pistol & Sword
as subclasses, and then we can use a SINGLE array of Weapon

```
public class Weapon{...}
public class Granade extends Weapon{...}
public class Pistol extends Weapon{...}
public class Sword extends Weapon{...}
Weapon[] weaponArr;
weaponArr[i] = new Granade();
weaponArr[j] = new Pistol();
weaponArr[k] = new Sword();
```

- In Game player's perspective, its not possible to anticipate how many
resources the player can collect at runtime.

Therefore, using array is not a
good option. Hence we can use one of the dynamic collection classes from
library which can grow on demand in phases.

- Let's decide that we are going to use ArrayList class to collect different
types of weapons for the player and demonstrate polymorphic behaviour of a
method (common-method or uncommon-method??? We are going to discover that)

2021-11-07 -- Scratchpad of CSE213 (Sec-1)
==========

- Abstract class & abstract methods
- Multiple Inheritance
  - Use of Interface

- Complete sample workflow for a IRAS goal
- Discussion on different UML diagrams which are typically used for system design.
  BUT due to time limitation, we will limit ourselves to "class-diagram" only.
  You will know more UML diagrams related to system design in
  "System Analysis and Design" core course in future

- Introduction to online tool "lucidchart" to create out class diagram

- How to use milestone-1 (CRA-report) to construct milestone-2 deliverables:
  - class diagram
  - database design (file system)

==========
Abstract class:

- It is a class which can't be instantiated/initialized
- We can only use it's handles to represent subclass instances

- For example, if we want to maintain an ARRAY of weapons within the context of a video game,
  where Weapon is a generalization but in reality only subclass instances exists within the
  memory, then we can use Weapon handles to instantiate Granade, Pistol and Swoed instances,
  provided that they are subclass of Weapon

```
public class Weapon{...}
public class Granade extends Weapon{...}
public class Pistol extends Weapon{...}
public class Sword extends Weapon{...}
```

```
                        Weapon[] weaponArr;
                        weaponArr[i] = new Granade();
                        weaponArr[j] = new Pistol();
                        weaponArr[k] = new Sword();
```

- In this example, Weapon qualifies to be an abstract class

- If there is NO abstract method in a class, still the author can declare the class as abstract. 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

Abstract Method:

- Abstract method is nothing but the UN-IMPLEMENTED method declared in superclass
- 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 implented(non-abstract) inherited method, but its optional. But it is mandatory for the subclass to override all non-implemented(abstract) inherited methods, so that of those methods are called using subclass instance as client, there exists some definition to execute.

Collection class:

- In your data structure course, you implemented a queue class

Ex of C++ collection class written by us:

```
                        class MyQueue{
                                private:
                                 //int vals[100];       // for fixed
sizes queue
                                 int *valPtr;           // for user
defined size
                                 //use of linked list/vector of ints
as the field, if size is unknown
                                 int front, back;
                        };
```

- Limitation: This queue can be used to enqueue/dequeue ONLY ints
- BUT, if we implement this MyQueue class as a GENERIC class, then we can use this MyQueue class instances to use as a queue of any datatype

Generic Collection class:

- stack, queue, list, vector, set, map, ... Array, ArrayList

- The collection classes that we use from library are typically implemented as generic collection class
- In C++, they come from STL (Standard Template Library)
- In Java, it also has its own generic collection classes which we are going to use extensively in our projects

Ex of Java generic collection class written by us:

```
                        public class MyGenericQueue<T>{
                                private T[] vals;       // for fixed
sizes queue
                                private int front, back;
                                public MyGenericQueue<T>(){
                                        Scanner s = new
Scanner(System.in);
                                        sout("Queue size? );
                                        vals = new T[s.nextInt()];
                                        front=back=-1;
                                }
                                public void enQueue(T val){...}
                                public T enQueue(){...}
                        }
                        public class Customer{....}
                        public class MainClass{
                           p s v main(...){
                                MyGenericQueue queueOfints<int> = new
MyGenericQueue<int>();
                                MyGenericQueue queueOfCustomers;
                                MyGenericQueue
queueOfCustomers<Customer>
                                        = new MyGenericQueue<Customer>
();
                           }
```

See netbeans for details:

```
    public static void main(String[] args) {
        //Weapon[] wArr = new Weapon[100];
        //Not a good idea as predicting array size is not realistic
idea
        //instead we can use a dynamic collection such as ArrayList
        ArrayList<Weapon> wList = new ArrayList<Weapon>();
        ...see netBeans
    }
```

Advanced 'for' loop:

- Regular for loop:
    - regular for loop can be used to access a collection

```
        for(initialValueOfLoopVariable; terminatingCondition;
howLoopVariableChanges){
                collectionName[loopVariableAsIndex]=...;
        }
```

    - regular for loop can ALSO be used just to iterate a block (Not using any collection)

```
        for(int i=1;i<100;i++) sout(i);
```

- Advanced for loop MUST be used to access a collection only

```
        for(typeOfCollectionElement collectionVariableNameToUse:
collectionName){
                collectionVariableNameToUse = .....; //for collection
of primitives
                collectionVariableNameToUse.doSome(); //for collection
of handles
        }
```

==========

Multiple Inheritance:

    - If a subclass has two or more (multiple) ancestors, then we call it multiple-inheritance

```
        public class Super1{
                //protected fields
                public void doSome1(){...}
                //public void print(){....}
        }

        public class Super2{
                //protected fields
                public void doSome2(){...}
                //public void print(){....}
        }
```

```
        public class Sub extends Super1, Super2{        //say for the
sake of discussion
                //additional private fields
                public void doSomeOwnWork(){...}
        }

        p s v m(....){
                Sub obj = new Sub();
                obj.doSome1();          //OK
                obj.doSome2();          //OK
                obj.doSomeOwnWork();    //OK
                //obj.print();          //Ambiguity, NOT OK
                //Two definitions of same signature is inherited from
two ancsetors
                //creating ambiguity/confusion to compiler, not
acceptable

                //C++ will leave the responsibility to discover such
ambiguity of multiple
                //definition conflict of same method inherited from
different superclasses
                //it will allow the subclass to have multiple
superclasses to acheive
                //multiple inheritance

                //On the other hand, Java knows that there is a
potential of having such definition
                //conflict in subclass, java does not ALLOW a subclass
to have more than one
                //superclass.
                Q: Then How multiple inheritance can be acheived in
Java?
                A:
                        - Java subclass can have one superclass as an
ancestor
                        - And the subclass can have rest of the
ancestors as "Interface"

Interface:
----------
        - It is a cousin of Abstract class
        - An interface can't have any implemented (non-abstract
methods). All methods os an
          intercace MUST be abstract method
        - Fields of an interface must be: static as well as final
(shared as well as constant)

================================================================
=================

'final' keyword:
-----------------
        - final field: constant, once value is given, it can't be
changed. value MUST be
          given at the time of memory allocation

                C++:    int x;
```

```
                        x=20;
                        const int y;    //error
                        const int y=20; //OK, y will be 20 for its
lifetime


                Java:   int x;
                        x=20;
                        final int y;    //error
                        final int y=20; //OK, y will be 20 for its
lifetime
```

- final method: its a superclass method (implemented), which can't be overridden in subclass

- final class: its a class which can't be extended