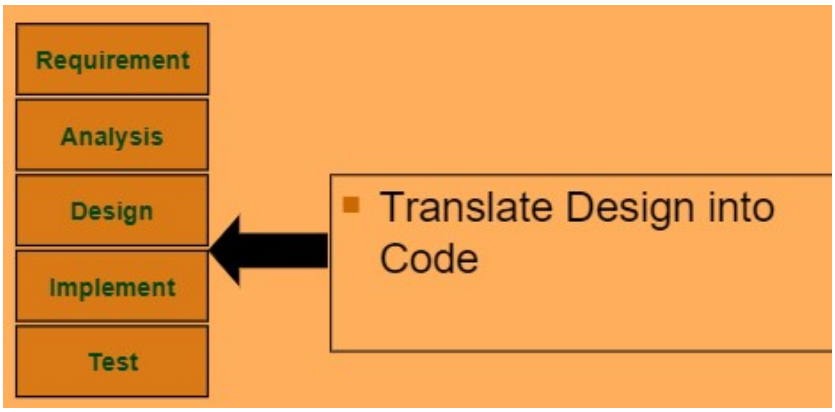
Lecture 10

Tuesday, March 21, 2023 12:03 PM

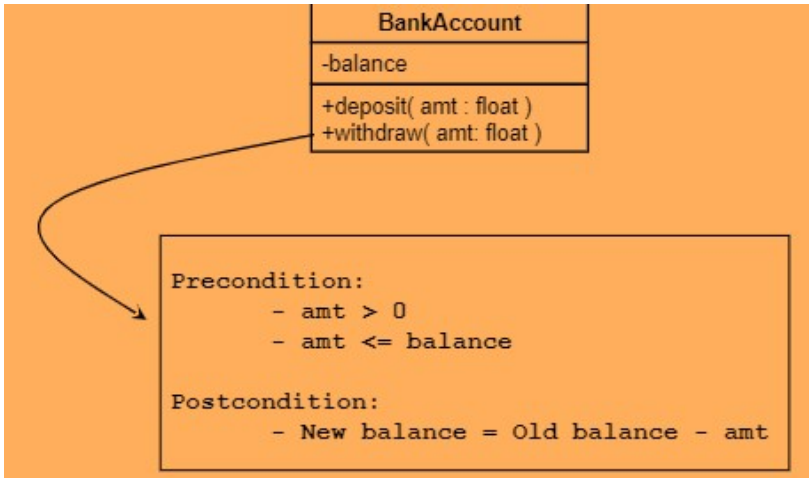
Overview of This Lecture

- Basic Implementation Steps.
- Implementing Classes:
  - Class Association:
    - Navigation direction;
    - Multiplicity constraint;
    - Qualified associations;
    - Association classes.

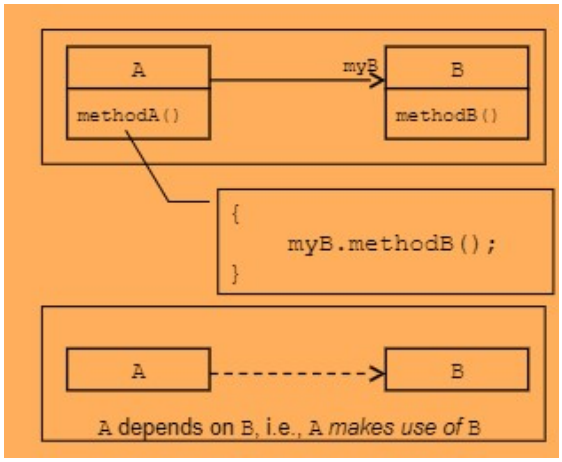
- 

- Implementation: Overview
  - Inputs:
    - Detailed Class Diagram
    - Interaction Diagram
    - Statechart
  - Activities:
    - Implementation of class diagram
  - Reminder:
    - Interaction diagrams show the basic behavior of some of the ops
    - Statechart describes a complete lifetime for classes with complex behavior
- Basic Steps in Implementation
  - Define method interface
    - Give a formal/informal description of the functionality of an operation
    - Useful in reducing misunderstanding
    - Lets splitting the workload
  - Decide order of implementation
    - Which class/subsys to start
  - Implement the design using a suitable pgrming lang
- Defining Method Interface
  - During design:
    - Detailed info abt the method header are defined:

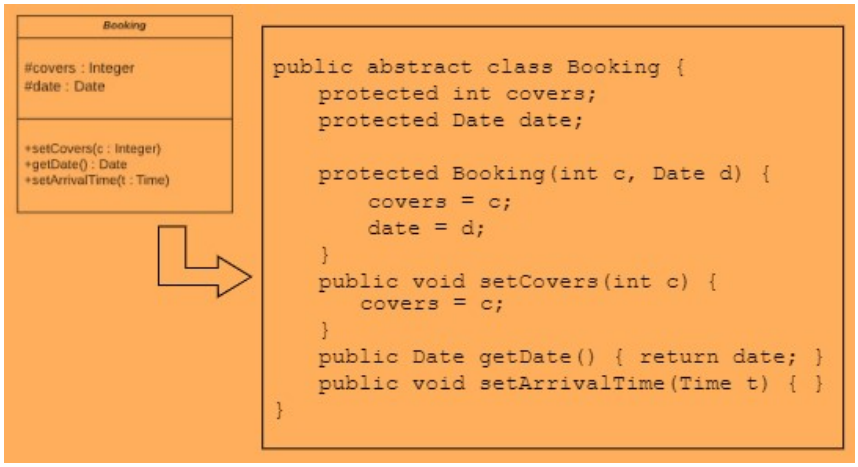
        - Returned Type, Parameter Type
        - Accessibility
    - But, functionality not formally defined:
        - What should method do?
        - What are acceptable vals for the parameter?
        - Etc..
    - Even more important for complicated operations
- Operation Contract
    - Writing the operation contract is 1 possible way to define functionality
    - Define precondition:
        - Express constraint abt attributes of the class and the actual parameter
    - Define Postcondition:
        - Express effect of an operation:
            - Instance creation/deletion;
            - Attribute modification;
            - Association Formed/Broken;
- Operation withdraw() Contract: EX:
    -
    -

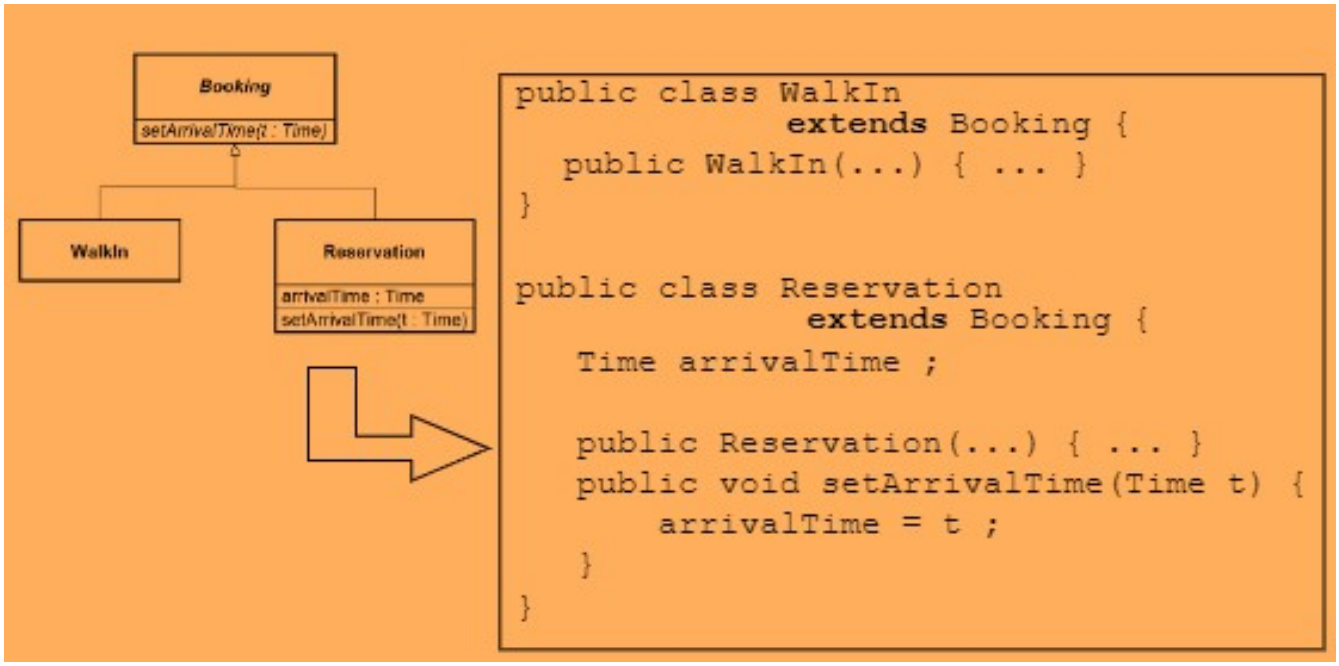


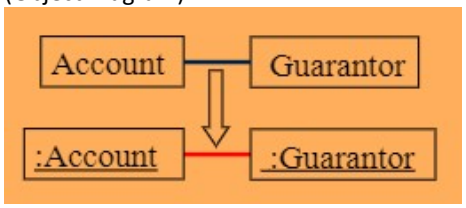
- Order of Implementation
    - Many classes w/ dependencies in the design:
        - Need an order of implementation
    - 2 basic strats:
        - Top-down:
            - Start w/ high-lvl components
            - Continue to implement the dependent components
        - Bottom-Up:
            - Start w/ lower lvl classes (like classes w/ least dependency)
            - Then go to code classes that make use of the implemented classes
- Ex:

- 

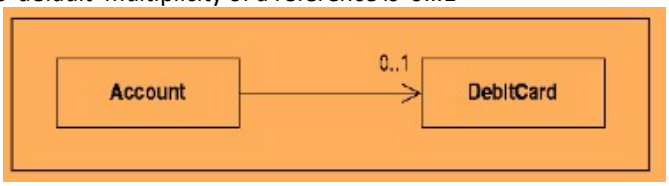
  - The myB can be an attribute in class A of class B;
  - The navigability is 1 possible indicator of the dependency
- Top Down Implementation
  - Starts w/ Class A, then Class B
  - Advantages:
    - Class A reps a *higher lvl* component as it makes use of other classes
    - Starting w/ high lvl components lets early validation of whole sys
  - Disadvantages:
    - Need temporary implementation (aka stub) for lower classes
    - Ex:
      - A temporary implementation of methodB() is needed for the implementation of methodA()
- Bottom Up Implementation
  - Starts w/ Class B, then Class A
  - Advantages:
    - Low lvl classes can usually stand alone
    - No need for for stub
  - Disadvantages:
    - Have to postpone the generation of a complete executable pgrm
- **Class Diagram Implementation**
- Implementing Classes
  - Basic mapping:
    - A UML Class is implemented by a class in Java, C++, etc.
    - Attributes are implemented by fields
    - Operations are implemented by methods
  - There can be some lang specific issues
    - Like the implementation of abstract classes, interfaces, and generalization
- EX: The Booking Class
  -

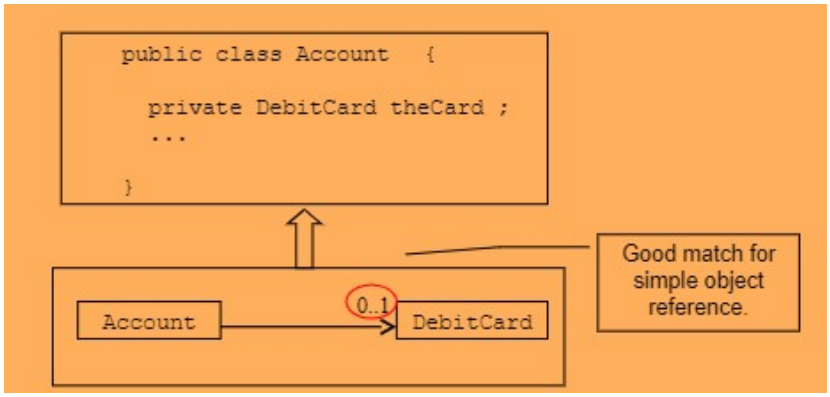
- 

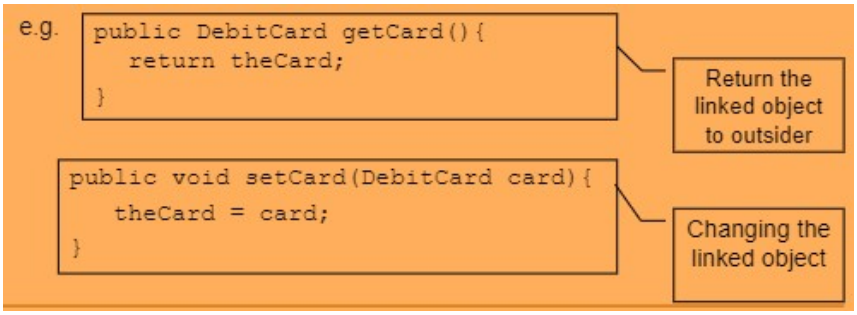
- Ex: Generalization
  - 

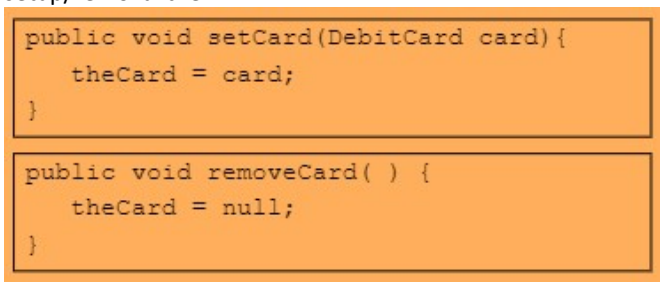
- Implementing Associations
  - Associations (Class Diagram) describe properties of links (Object Diagram)
  - 

  - A link gives one object access to another and lets message passing
  - Java: References (C++: Object Pointer) share these properties, so associations can be implemented w/ references
  - References support only 1 direction of navigation
    - Should restrict navigability
  - References should not be manipulated explicitly by other classes
- Optional Unidirectional Association
  - Distinction btwn associations
    - Direction
    - Multiplicity
  - Reference vars can hold

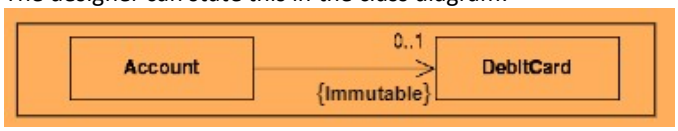
        - A reference to another object
        - Or the null reference
    - So the 'default' multiplicity of a reference is '0...1'
    - 

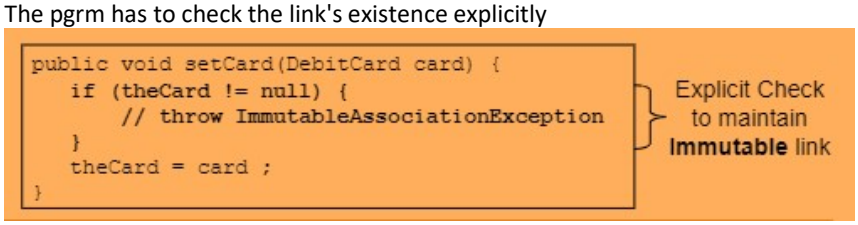
- Implementation
    - The association is defined by a field in the Account Java class holding a reference to an instance of the DebitCard Java class:
    - 

- Maintaining the Association
    - Account is "aware" of this association, it should also maintain the association:
        - Setup/remove the link;
        - Appropriate methods for accessing the linked object
        - 

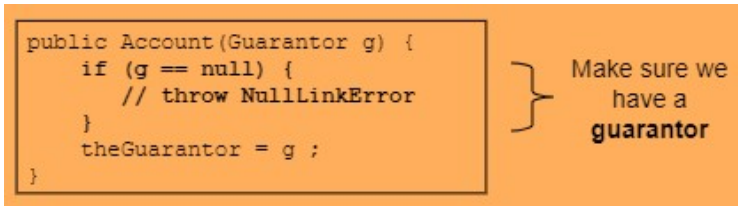
- Mutability for Association
    - Last example assumes that the DebitCard association is mutable:
        - An Account object can be linked to a diff DebitCard object during its lifetime
    - Hence, it makes sense to have the following methods for setup/removal the link:
    - 

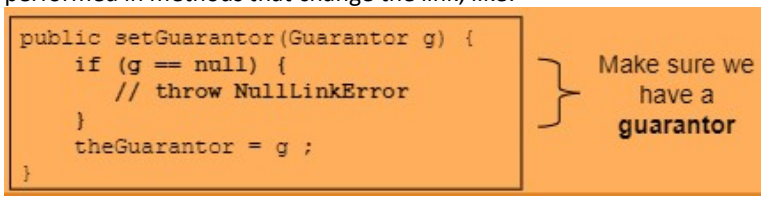
- Immutable Association
    - Some associations are immutable:
        - Once assigned, a link must not be altered

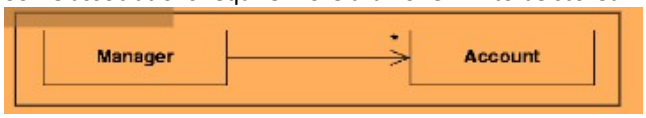
- The designer can state this in the class diagram:
- 

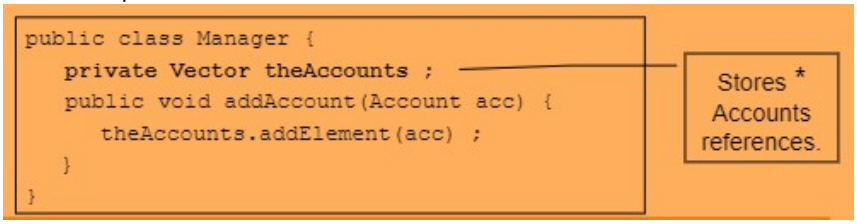
- The pgrm has to check the link's existence explicitly
- 

- Compulsory Association
  - Multiplicity w/ lowerbound larger than 0 is compulsory, like "At all times, the link(s) cannot be null"
  - Ex:
  - 

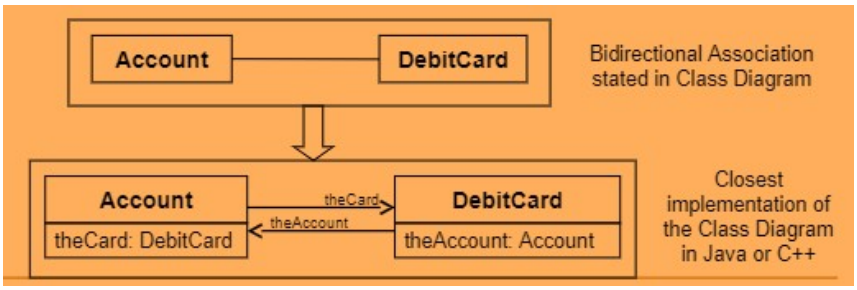
  - That ^ means an "Account object must have exactly one link to a Guarantor object at all times"
  - Pgrmer have to check explicitly to ensure this is true
- Implementation
  - The earliest possible opportunity to setup the link is in the constructor method:
  - 

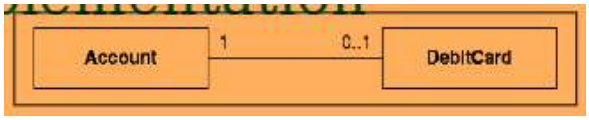
  - If the association is mutable, similar checks must be performed in methods that change the link, like:
  - 

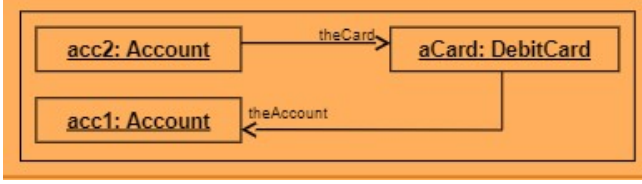
- 1-to-Many Unidirectional Association
  - Some associations require more than one link to be stored:
  - 

  - Data structures, like arrays and vectors, can be used to store the references
    - For a specific upper bound, like "0...8", specific checks must be implemented:
    - 

- Bidirectional Association
  - Some associations need to be navigated in both directions:
    - Bc references are unidirectional, it will take two

references to implement a bidirectional link
    - The association can be implemented by including a suitable field in each of the associated classes
    - 

- Bidirectional Association Implementation
    -
    - 

    - DebitCard is optional ("0...1" from Account)
    - Account is compulsory ("1" from DebitCard)
    - Hence, best way to satisfy the conditions is
        - Make the Account first, then the DebitCard:
        -
```
{
   Account acc1 = new Account() ;
   DebitCard aCard = new DebitCard(acc1) ;

   acc1.setCard(aCard) ;
}
```
- Referential Integrity
    - The bidirectional association is implemented as 2 separate references, its important to keep them consistent
        - Links should be "inversed" one to each other
        - Known as referential integrity
    - Ex of referential integrity violation:
        -
```
Account acc1 = new Account() ;
Account acc2 = new Account() ;
DebitCard aCard = new DebitCard(acc1);
acc2.setCard(aCard) ;
```


***

- Maintaining Referential Integrity
    - Prob w/ ex code ^:'
        - DebitCard making and reference setup (setCard() method) are two separate operations
        - It's the responsibility of pgrmer to make sure the 2 ops are performed in a consistent fashion
    - To min human error, delegate one of the classes to keep the referential integrity instead
    - Ex:

-

```
public class Account {
   private DebitCard theCard ;

   public void addCard() {
           theCard = new
    DebitCard(this);
   }
   // other methods as well
}
```

Account class handles both the DebitCard creation and reference setting.

- Joint Creation of Objects
  -
  - 

  - Suppose an association is intended to be immutable and also needs to be traversed in both directions
  - Constructors of both classes should take in an instance of the other to satisfy the constraint specified by the class diagram
    - Java and C++ do not allow simultaneous object creation
    - Must create one of the objects with the default constructor, like to violate the constraint for a short time
- Code Ex:
  - Attempts to satisfy the compulsory link:
  -

```
Account a = new Account(g) ;
Guarantor g = new Guarantor(a) ;
```

g is **not** initialized.

  -

```
Account a = new Account(new Guarantor(a)) ;
Guarantor g = a.getGuarantor( ) ;
```

a is **not** initialized.

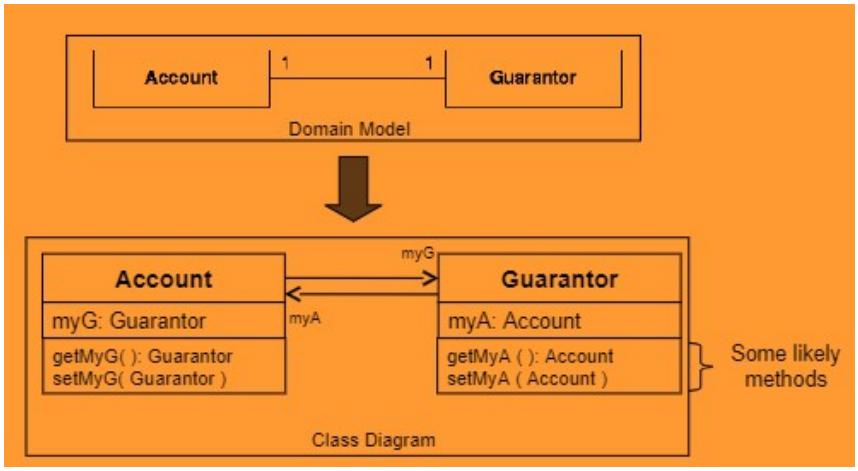
  - Have to violate the constraint for a while:
  -

```
Guarantor g = new Guarantor( ) ;
Account a = new Account(g) ;

g.setAccount(a) ;
```

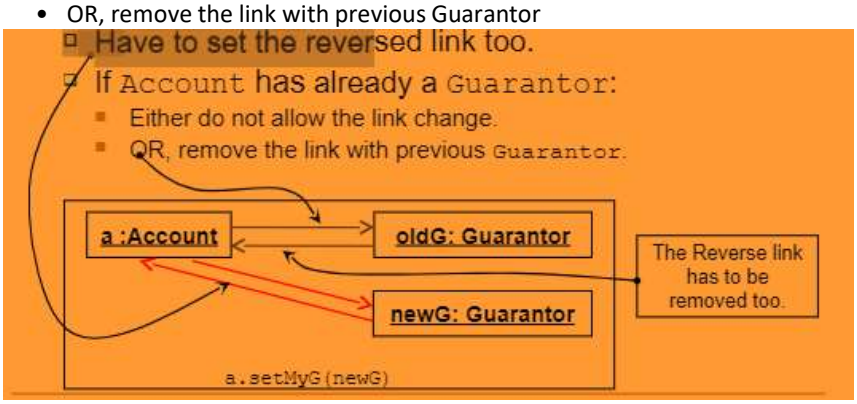
g is initialized by the default constructor without an Account reference.

Setup the reference after Account a is properly initialized.

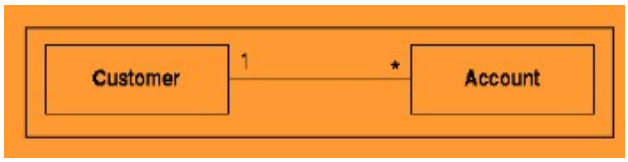
- Implementing Bidirectional Association
-



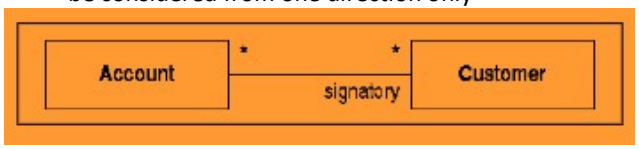
- Maintaining Bidirectional Association

- Methods that change the reference should maintain referential integrity
- Take setMyG() for example:
  - Have to set the reversed link too
  - If Account has already a Guarantor:
    - Either do not allow the link to change
    - OR, remove the link with previous Guarantor
  - 

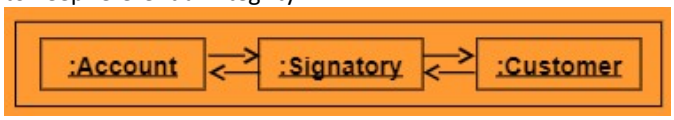
- Code Ex:
  - ```
    //Class Account
    public void setMyG(Guarantor newG) {
      if (newG == null)
         //throw NullLinkError

      if (myG != null && myG != newG) {
         //previous Guarantor exists
         myG.removeMyA(this);
      }

      if (myG == newG)
         return;

      myG = newG;

      //set the reverse link
      myG.setMyA(this);
    }
    ```
  - ```
    //Class Guarantor
    void removeMyA(Account a) {

       //only allows holder of
       //the reverse link call
       //this method

       if (a == myA)
           myA = null;
    }
    ```
  - ```
    //Class Guarantor
    public void setMyA(Account newA) {
      if (newA == null)
         //throw NullLinkError

      if (myA != null && myA != newA) {
         //previous Guarantor exists
         myA.removeMyG(this);
      }

      if (myA == newA)
         return;

      myA = newA;

      //set the reverse link
      myA.setMyG(this);
    }
    ```
  - ```
    //Class Account
    void removeMyG(Guarantor g) {

       //only allows holder of
       //the reverse link call
       //this method

       if (g == myG)
           myG = null;
    }
    ```
- Observations
  - Even w/ the complicated code, there are still some problems:
    - The old guarantor will be without an Account object
    - Mutual recursive functions:
      - The setMyA calls setMyG, which callse setMyA

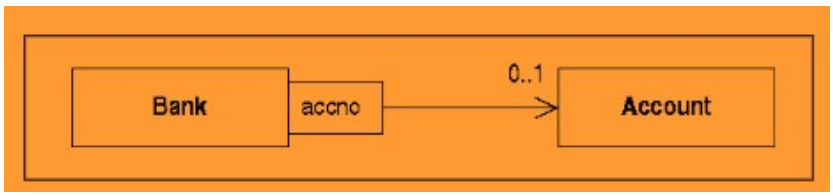
        - Easy to get into infinite loop
    - Should avoid compulsory bidirectional association if possible
    - If a bidirectional association must be used, immutability is recommended to curb the coding complexity
- One to Many Association
    -
    - 

    - Rais no new issues:
        - Customer holds a collection of references
        - Account holds a non-null reference
    - Customer is a better choice as the "maintainer" for the referential intergrity:
        - It gives a method like addAccount() that makes a new account and setup the reference
        - Similar to the addCard() method b4 ***
- Many-to-Many Association
    -
    - 

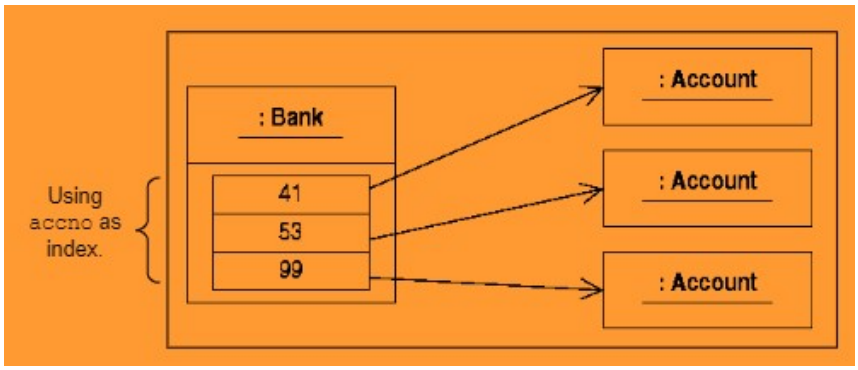
    - Previously discussed implementation techniques can be used
    - Recommendation
        - Lets only 1 of the classes to modify the links
        - Like it makes sense to allow only the Customer to change links to Account and keep the reversed link also
        - Simplify the implementation as the coding logic can be considered from one direction only
    - 

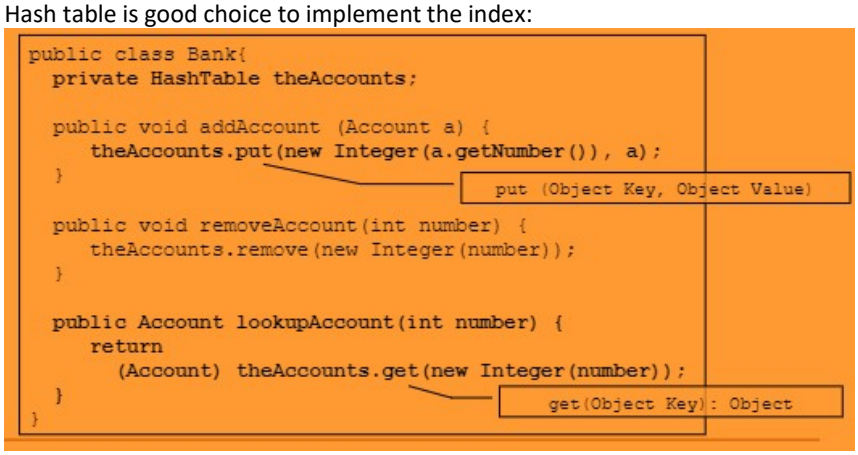
    - Another implementation technique is to reify the association
    - 

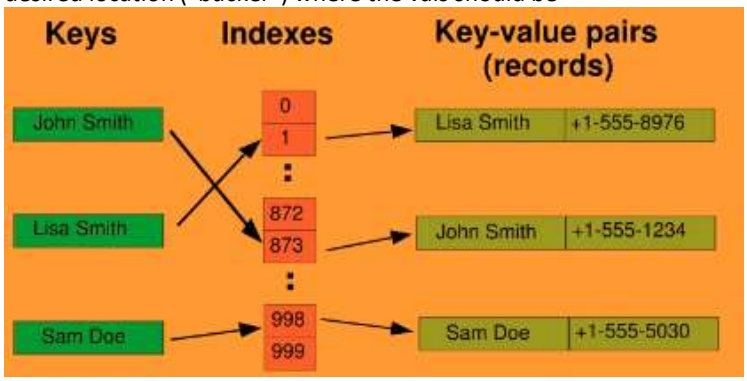
    - The new class (like Signatory like above) is in a good position to keep referential integrity
    - 

- Qualified Association
    -
    - 

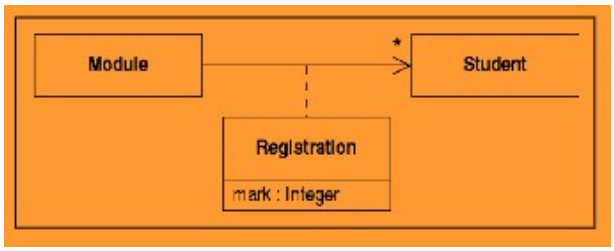
    - Purpose of a qualifier is often to give efficient access to

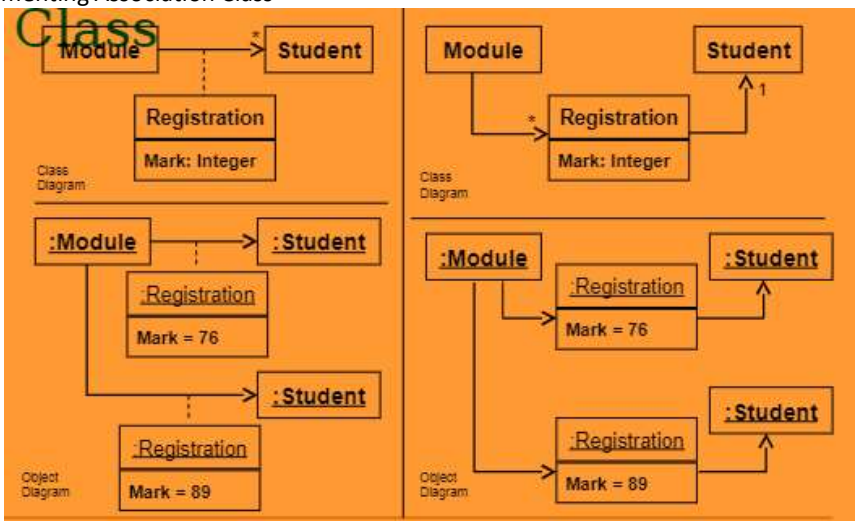
linked objects:
    - Ex: to access accounts given only the account number
    - Also to avoid a linear search through all accounts
- Implementing Qualifier
    - The run-time structure involves some kind of *index* to accounts:
    - 

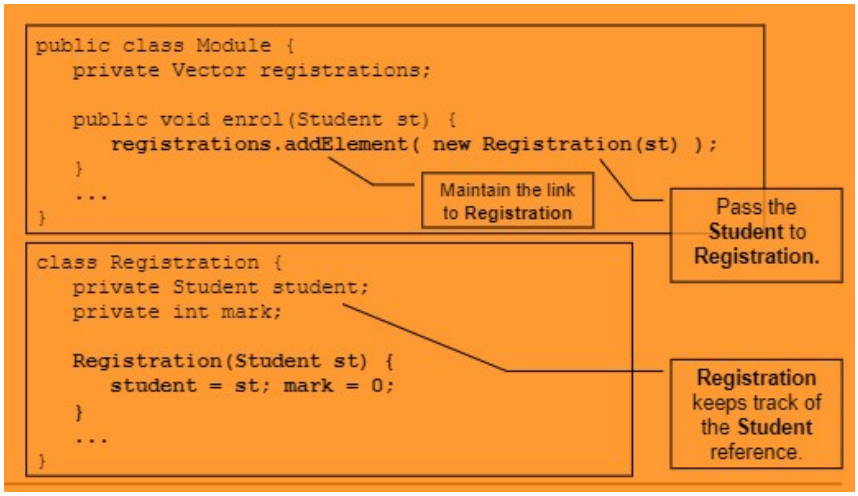
- Code Ex:
    - Hash table is good choice to implement the index:
    - 

- Hash table – reminder
    - Means a data structure that associates keys w/ vals
    - Primary op it supports efficiently is a lookup: given a key (like person's name), find the corresponding value (like the person's telephone number)
    - Works by transforming the key using a hash fn into a *hash*, a number that is used to index into an array to locate the desired location ("bucker") where the vals should be
    - 

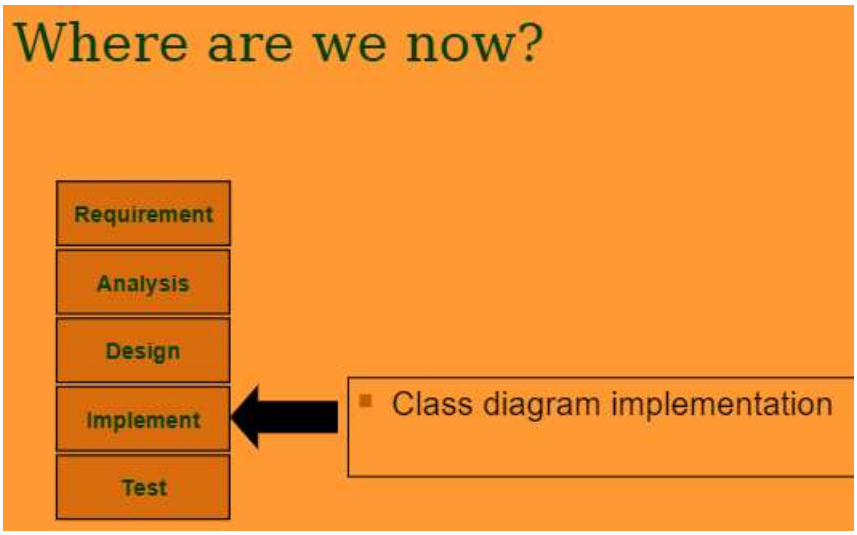
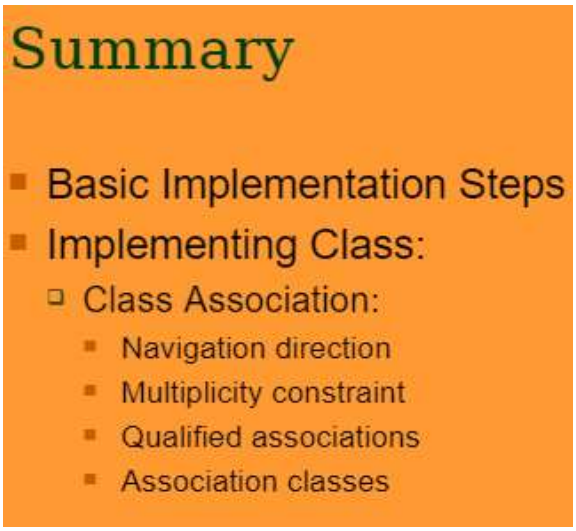
    - Like arrays, hash table gives constant-time O(1) lookup on average, regardless of numb of items in the table
- Association Class
    -

- 

  - Common strat:
    - Transform the association class into a simple class, replacing it by 2 associations
- Implementing Association Class
- 

- Code Ex:
- 

-

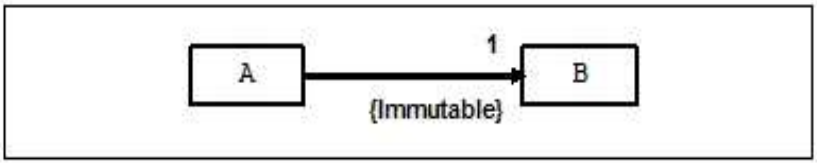
- 
Where are we now?
Requirement
Analysis
Design
Implement
Test
Class diagram implementation
- 
Summary
Basic Implementation Steps
Implementing Class:
Class Association:
Navigation direction
Multiplicity constraint
Qualified associations
Association classes
-
  -

L10 Assessment

Tuesday, March 21, 2023 9:35 PM

Question 1
10 Points

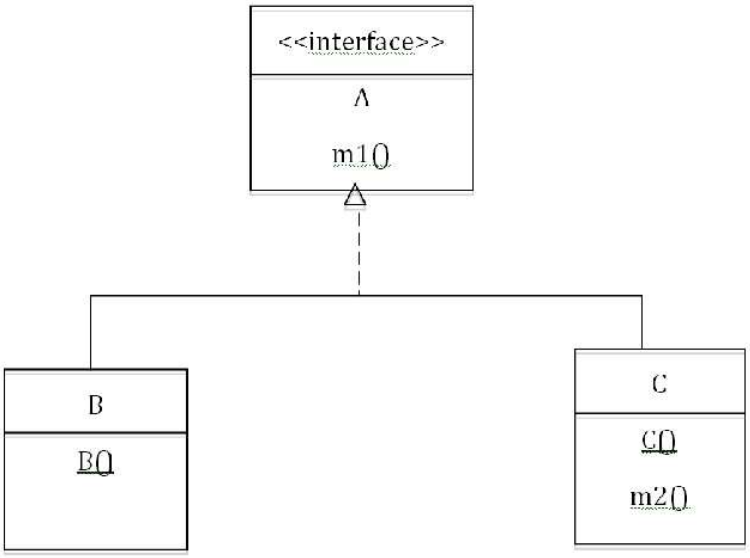
Given the below UML class diagram, write a simple Java program that implements it. Include in class A a constructor and access methods for the object of class B referred in class A.




```
13   * This is Q1 on L10
14   * */
15  public abstract class A{
16      private B objectB;
17      private A(B b) {
18          if(b == null) {
19              //throw NullLinkError
20          }
21          objectB = b;
22      }
23      public void setB(B b) {
24          if(objectB != null) {
25              //throw ImmutableAssociationException
26          }
27          objectB = b;
28      }
29  }
30  public final class B{
31      protected B() {
32          //...
33      }
34  }
```

Question 2
10 Points

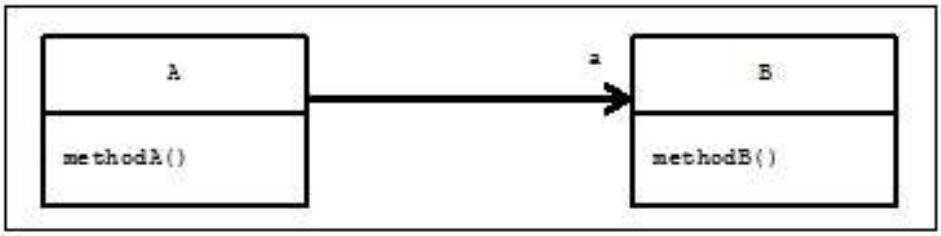
Implement in Java/C++ language the below UML class diagram:




```
//the <<interface>> had no class name
//so I assumed it was A
interface A{
    void m1();
}
class B implements A{
    public void m1() {
        //...
    }
    B(){
        //...
    }
}
class C implements A{
    public void m1() {
        //...
    }
    C(){
        //...
    }
    void m2() {
        //...
    }
}
```

Question 3
10 Points

What is true when doing the implementation of the following class diagram?



The implementation of A may contain an attribute a of class B;
The implementation of B should contain an attribute a of class A;
The implementation of methodB() should contain the reference a;
Class B makes use of class A;
None of the above.

Question 4
10 Points

What is true regarding the top down and bottom up implementations?

Top down strategies must postpone the generation of complete executable program;
High level classes can usually stand alone;
Bottom up strategies need temporary stubs for lower classes;
Starting with a high level component enables early system testing;
None of the above.