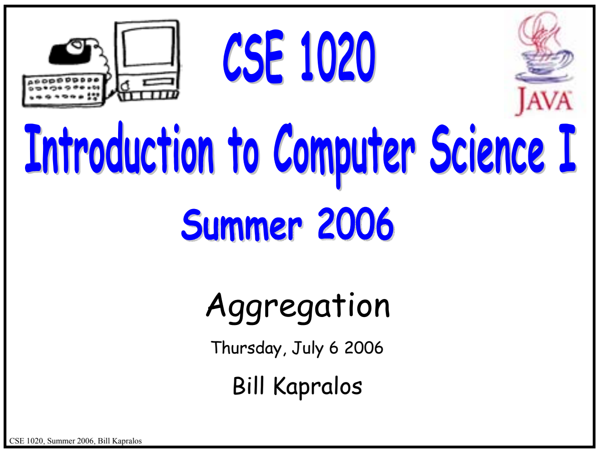
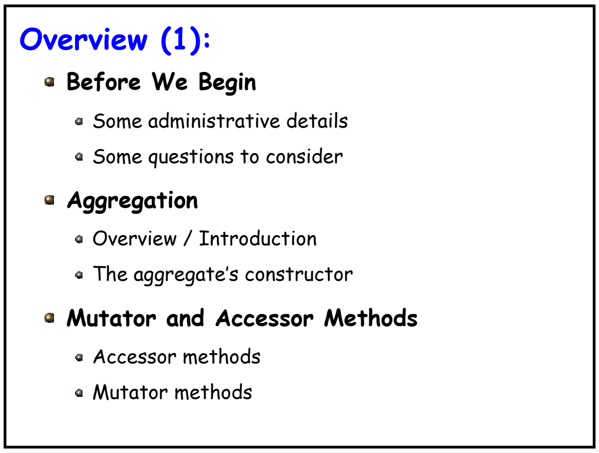


CSE 1020
Introduction to Computer Science I
Summer 2006

Aggregation

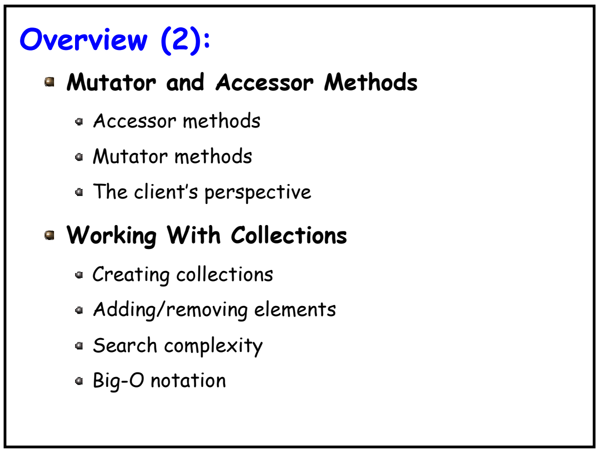
Thursday, July 6 2006

Bill Kapralos



Overview (1):

- **Before We Begin**
  - Some administrative details
  - Some questions to consider
- **Aggregation**
  - Overview / Introduction
  - The aggregate's constructor
- **Mutator and Accessor Methods**
  - Accessor methods
  - Mutator methods



Overview (2):

- **Mutator and Accessor Methods**
  - Accessor methods
  - Mutator methods
  - The client's perspective
- **Working With Collections**
  - Creating collections
  - Adding/removing elements
  - Search complexity
  - Big-O notation

Before We Begin

Administrative Details (1):

- **Lab Test 2**
  - Due to technical problem, lab test 2 will be re-scheduled during next week's lab period

Some Questions to Consider (1):

- What is debugging ?
- What are the four main debugging techniques ?
- Describe the use of print statements to debug code
- When describing a class in UML, what are the three possibilities we examined ?
- What are the three relationships we looked at in UML ?
- What is aggregation ?
- What is inheritance ?

Aggregation

Overview (1):

- **What is Aggregation ?**
  - The classes used by an application we develop may contain attributes that themselves are instances of other classes → objects within objects
    - This is known as aggregation
  - The presence of aggregation makes the features of both the aggregate and the aggregated available directly or indirectly to the program
  - Aggregation is the most general framework for implementing layering
    - We will learn how to recognize aggregates , what to expect to see in their APIs…

Introduction (1):

- ***Aggregation and Real-World Analogies***
  - The world we live in is full of "complex" objects → objects made up of other objects
    - A car is a compound object that contains a radio → the radio itself is a separate object manufactured at a different factory
    - A portable CD player that plays CDs → a CD that it plays is itself an object
  - We will need to determine whether an object is compound or not
    - Everyday objects exhibit a "whole-part relationship" → the same holds with software

Introduction (2):

- **Some Definitions**
  - Aggregate
    - A class C is an aggregate if one of its attributes (instance variables) is an object reference
    - Assuming the reference is of type T → C and T have a "has-a" relationship and C (the "whole") has a T (the "part")
    - This relationship between C and T is also called aggregation
    - If the attributes of a class are all primitive → the class is not an aggregate

Introduction (3):

- **Some Definitions (cont.)**
  - Aggregate (cont.)
    - But what about Strings that are masqueraded as primitive types → this leads to a revised definition of aggregation
  - Revised definition of aggregation
    - An aggregate class has at least one attribute whose type is not primitive or String

Introduction (4):

- **Aggregation Example**
  - Suppose we create an instance (object) of class C
    - This involves assigning values to all attributes of C including the object reference of type T
    - Assigning a value to the object T means to create an instance (object) of type T and assign the reference to it (of course we can also assign the value null to the reference but we will ignore this)
    - Every instance of object C has an instance of T within it → an object within an object e.g., "whole-part"

Introduction (5):

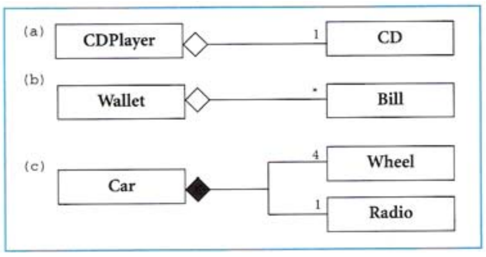
- **Aggregation and UML**
  - An example
    - Aggregation between CDPlayer class and CD class → CDPlayer class has one attribute of type CD
    - Multiplicity → the number of attributes of a particular class and it is written next to the aggregated part
    - Variable multiplicity → the number of attributes of a particular class is not fixed and the aggregate class is known as the collection and each instance is known as an element of the collection

Introduction (6):

- **Aggregation and UML (cont.)**
  - An example (cont.)
    - A class may aggregate several classes, each with a different multiplicity
    - Composition → if creating an instance of the aggregate automatically leads to creating an instance of an aggregated part, the aggregation then becomes a composition (the aggregate and the aggregated object are born together and die together)

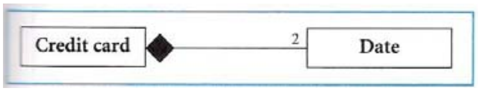
Introduction (7):

- **Aggregation and UML (cont.)**
  - An example → graphical illustration



Introduction (8):

- **Aggregation and UML (cont.)**
  - Another example → CreditCard class in type.lib is an example of an aggregate
    - Encapsulates a credit card and has the following attributes → card number, cardholder's name, credit limit, balance, issue/expiry dates
    - Aggregate because issue/expiry dates are of the class type Date (java.util) → actually a composition since automatically created/deleted



The Aggregate's Constructor (1):

- **Attributes Initialized in Constructor**
  - When we create an instance of aggregate *C*, the constructor of *C* is used to initialize *C*'s attributes
    - Since one of the attributes is an object reference, it is initialized by making it point to an instance of the aggregate T
    - But who instantiates T → two possibilities

      1. The aggregate's constructor
      2. The client

The Aggregate's Constructor (2):

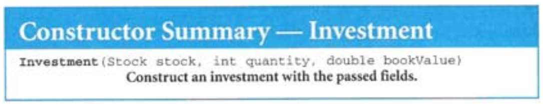
- **Attributes Initialized in Constructor (cont.)**
  - Initializing in the aggregate's constructor
    - As a client, we do not do anything with T → create *C* and this automatically creates T and we don't necessarily need to be concerned with it
    - Corresponds to a composition → the part is created together with the whole by the whole and since we don't have access to it, it dies when the whole dies

The Aggregate's Constructor (3):

- **Attributes Initialized in Constructor (cont.)**
  - Initializing in the client
    - We create (in the client) the part instance and pass its reference to the constructor of the aggregate as an argument
    - The part is created before the whole
    - Since the reference is created in the client, we have access to the reference in the client and it can remain alive even if the whole dies → not a composition

The Aggregate's Constructor (4):

- **Attributes Initialized in Constructor (cont.)**
  - Example → Investment class
    - Consider the constructor shown below



Constructor Summary — Investment

`Investment(Stock stock, int quantity, double bookValue)`
Construct an investment with the passed fields.

  - Constructor must receive a reference of type Stock from the caller
    - In other words, the Stock object must be created in the client and passed to the Investment constructor

The Aggregate's Constructor (5):

- **Attributes Initialized in Constructor (cont.)**
  - Example → Investment class
    - Consider the following code fragment

```
int number = 15;
double cost = 12.25;
Stock stock = new Stock(".AB");
Investment inv = new Investment(stock, number, cost);
```

The Aggregate's Constructor (6):

- **Attributes Initialized in Constructor (cont.)**
  - Example → Investment class (cont.)
    - stock and inv don't share a common lifetime and inv can be deleted without deleting stock

```
Stock stock = new Stock(".AB");
{
   Investment inv = new Investment(stock, number, cost);
}
Output.println(stock);
```

    - At the end of the fragment, the "whole" is deleted (exit its scope), while the "part" remains

Mutator and Accessor Methods

Accessor Methods (1):

- **What is an Accessor Method ?**
  - Allows a user of the class to access data from the class (e.g., instance variables etc.) that may not be directly accessible (e.g., may be private)
    - Again, the user does not need to be aware of how the data is actually maintained by the class → the user only needs to know the value!
    - Typically these methods do not have any parameters

```
public int getWidth()
```

Accessor Methods (2):

- **What is an Accessor Method ? (cont.)**
  - Example → Consider the area attribute of a Rectangle
    - Can have instance variable that holds the area value or,
    - When needed, it can be computed "on the fly" (e.g., width × height)
    - Whether or not we have an instance variable or not, is irrelevant to the user of the class → this is an implementation detail

Accessor Methods (3):

- **Privacy Leaks**
  - The returned reference of an accessor method may point directly at the aggregated part itself or at a copy of it
    - If it is pointing to the part itself, then the client (the one who invoked the accessor) has access to the part directly and can actually change the state of the part → this is known as a privacy leak
  - Which approach does the Investment class follow ?
    - Lets find out…

Accessor Methods (4):

- **Privacy Leaks (cont.)**
  - To determine the approach the Investment class uses, consider the following code segment

```
Stock stock = inv.getStock();
boolean old = stock.titleCaseName;
stock.titleCaseName = !old;
boolean isCopy = (inv.getStock().titleCaseName == old);
System.out.println(isCopy);
```

  - After executing above code fragment
    - isCopy == false → reference returned
    - isCopy == true → copy returned

Accessor Methods (5):

- **Privacy Leaks (cont.)**
  - Consider the following code segment

```
Person citizen = new Person("Joe Citizen",
    new Date("January", 1, 1900), new Date("January", 1, 1990));
Date myDate = citizen.getBirthDate();
myDate.setDate("April", 1, 3000);
```

  - We have basically circumvented the private declared instance variable
    - We have changed the birth date of the Person object to a date after the death → impossible!

Accessor Methods (6):

- **Privacy Leaks (cont.)**
  - But myDate refers to the private instance variable! → we have changed the value of a private declared instance variable!
    - When returning a class type instance variable, be sure to avoid this memory leak
    - Return a copy of the class - not the reference!

Mutator Methods (1):

- **Typically Class Attributes (Instance Variables) Will be Private**
  - Do not have direct access to the attributes
    - But there may be times that we need to change the value of data within an object
  - But allowing the programmer to access an objects data (e.g., declaring instance variables as public) goes against the whole idea of information hiding
    - This means the user of our class is aware of the implementation details!

Mutator Methods (3):

- **What is a Mutator Method ?**
  - A public declared method of a class
    - Allows the user of the class to modify attributes of an object without having to worry about the underlying implementation and without access to the data (instance variables)
    - User passes the new data via the mutator method and the method changes the data regardless of the implementation
    - Also allows for a check to be made to ensure that the passed data is valid → this may not be the case if the user had access to the data.

Mutator Methods (4):

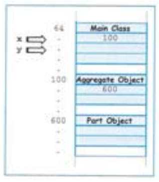
- **What is a Mutator Method ? (cont.)**
  - Typically, mutator methods begin with the word "set" to indicate it is a mutator method
  - Many times they also do not return a value → "void" method
    - At times they can return a boolean → true is returned if the passed argument is valid and false otherwise

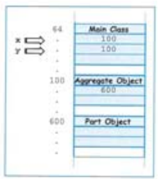
```
public void setWidth(int width)
```

Mutator Methods (5):

- ***Aggregate Cloning***
  - Given instance x of an aggregate, make a copy y
  - Three ways to perform the copying
    - Aliasing (set y equal to x) → a second reference and the instance itself is not copied and any changes made through x are reflected through y

Before the copy

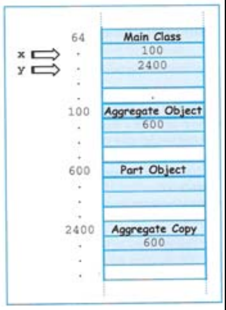




After the copy

Mutator Methods (6):

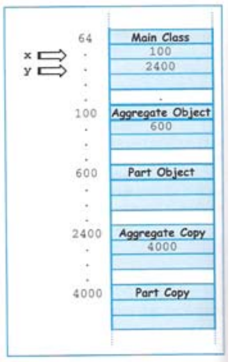
- **Aggregate Cloning (cont.)**
  - Three ways to perform the copying (cont.)
    - Shallow copying → make a new instance of the aggregate with same state as the given one but of course, non-primitive attributes are simply aliased and not copied (any changes made through x will be reflected through y)



After shallow copying

Mutator Methods (7):

- **Aggregate Cloning (cont.)**
  - Three ways to perform the copying (cont.)
    - Deep copy → y will point to a new instance in which each non-primitive attribute is itself deep-copied and any changes made throughx will not be seen through y



After deep copying

The Client's Perspective (1):

- **Aggregation and Layered Abstraction**
  - As a client, we delegate work to other components
    - Implementation of the components does not have to be from "scratch" → they can also delegate their work to other components
    - This second layer of delegation occurs when the implementer uses a feature of another class → create instance of that class and store it as an attribute
  - But then this implies that aggregation is an implementer's concern ???

The Client's Perspective (2):

- **Aggregation and Layered Abstraction (cont.)**
  - But then this implies that aggregation is an implementer's concern ???
    - Why should the client care if implementer performs the task from scratch or uses other components → doesn't knowing this information lead to "breaking the encapsulation" ???
    - Flexibility → at one extreme components are so basic clients find them complex to build an application out of them and at the other extreme, components are too complex and the client can't find components to meet its requirements

The Client's Perspective (3):

- **Aggregation and Layered Abstraction (cont.)**
  - Aggregation provides a solution that combines the benefits of both extremes
    - "Basic" components are still able to interface with other components
    - Gives client flexibility in choosing components without the complexity of assembling them
  - Consider a real-world analogy
    - Building a car with what may be a peculiar combination of features…

Working With Collections

Recall (1):

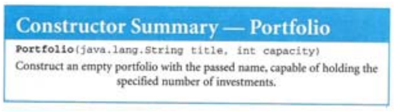
- **What is a Collection**
  - An aggregate whose multiplicity is variable and which the aggregated parts are called elements
  - A variable amount of variables, requires the use of a special API
    - Enable the client to add and remove elements on demand, browse elements or search for a particular element

Creating the Collection (1):

- **Collection Constructor**
  - The elements of the collection don't have to be specified upon creation
    - Basically, the constructor creates an empty collection
    - Other methods allow us to add elements after the collection has been created
  - How much memory is set aside for the collection ?
    - Static allocation sets aside memory once (according to application) and doesn't change afterwards → any potential problems with this ?

Creating the Collection (2):

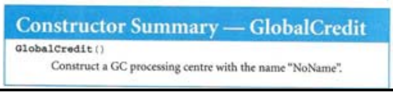
- **Collection Constructor (cont.)**
  - How much memory is set aside for the collection ?
    - Dynamic allocation sets aside memory "on the fly" as it is needed → application doesn't need to specify an initial size
  - Lets look at the constructor of the Portfolio collection



Constructor Summary — Portfolio

`Portfolio(java.lang.String title, int capacity)`

Construct an empty portfolio with the passed name, capable of holding the specified number of investments.



Constructor Summary — GlobalCredit

`GlobalCredit()`

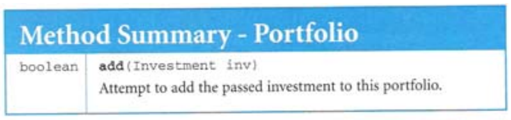
Construct a GC processing centre with the name "NoName".

Adding/Removing Elements (1):

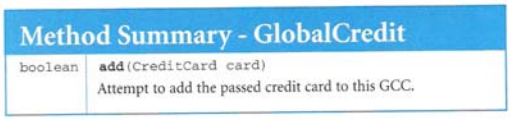
- **Adding Elements**
  - A collection must provide methods for inserting elements → often called add
  - Two issues can arise
    - Collection is full → statically allocated methods that cannot grow automatically - add method fails
    - Element is already present → trying to insert the same element twice (by "same" we mean the equals() method returns true) although some collections do allow for duplicate elements

Adding/Removing Elements (2):

- **Adding Elements (cont.)**
  - Add method in the Portfolio class



Method Summary - Portfolio	
boolean	add(Investment inv) Attempt to add the passed investment to this portfolio.



Method Summary - GlobalCredit	
boolean	add(CreditCard card) Attempt to add the passed credit card to this GCC.

Examples (1):

- **What is the Easiest Way to Learn About Collections ?**
  - Practice!
    - Lets examine some code → Figures 8.15 – 8.20

Search Complexity (1):

- **Defining Search Complexity**
  - The number of tests it must perform in the worst case before it can reach a conclusion → e.g., determine whether element is or is not in the list
    - If a collection has N elements complexity of exhaustive search is N → why ?
  - Complexity of algorithms/programs including search is given in big-O notation → exhaustive search has a complexity of O(N)
    - Since it is a linear function of the number of elements in the collection → linear search

Complexity and big-O (2):

- **Complexity Doesn't Measure Execution Time**
  - Provides a measure of how the execution time depends on the size of the input
    - Consider linear search with O(N) complexity → if we double the input then we double the complexity (e.g., leads to a doubling of execution time)
  - Some different big-O complexities
    - $O(1)$, $O(\log N)$, $O(N \log N)$, $O(N^2)$, $O(N^3)$, $O(N!)$ …
  - Complexity is concerned with large input not small!
    - Not concerned with complexity for small N

Complexity and big-O (3):

- **Big-O**
  - For large values of N it can be shown that the execution time T of a program with complexity O(f(N)) is given by

    $T \approx \alpha f(N)$

    - where α is a proportionality constant → allows us to predict execution time at N_1 given its value at N_2