Relationships between Objects

Object Equality
Composition and Aggregation

COMP2603
Object Oriented Programming 1

Week 3 Lecture 2

Outline

- Types of Relationships in Object-Oriented Programming
  - Aggregation
  - Composition
- Implementing Relationships
  - Aggregation
  - Composition
- Object Equality

Object Class

Recall, some of the methods inherited from the Object class are:

- `public boolean equals(Object obj)`
  Indicates whether some other object is "equal to" this one.
- `public int hashCode()`
  Returns a hash code value for the object.
- `public String toString()`
  Returns a string representation of the object.

Object Equality

Consider the equals( ) method of the Object class:

`public boolean equals(Object obj)`

This method returns true if the current object is stored at the same memory address as `obj` and false otherwise.

The method behaves just like = =

Example 1

```
int x = 10;
int w = 20;
if (x == w)// checks value at memory location
    System.out.println(“variables are equal”);
else
    System.out.println(“variable are not equal”);
```

Output: “variables are not equal”

Example 2

```
String s1 = “abc";
String s2 = “def”;
if (s1 == s2) // checks memory location
    System.out.println(“strings are equal”);
else
    System.out.println(“strings are not equal”);
```

Output: “strings are not equal”

Example 3

```
String s1 = “abc";
String s2 = “abc”;
if (s1 == s2) // checks memory location
    System.out.println(“strings are equal”);
else
    System.out.println(“strings are not equal”);
```

Output: “strings are not equal”

✗ **But this output could be conceptually incorrect**

Example 4

```
String s1 = “abc";
String s2 = “abc”;
if (s1.equals(s2))
    System.out.println(“strings are equal”);
else
    System.out.println(“strings are not equal”);
```

Output: “strings are equal”

Why? Because the String class provides its own equal method

Aggregation

An aggregation relationship models a whole-part relationship between two classes. It is sometimes referred to as a part-of relationship.

Consider two classes, A and B. An instance of A (referred to as the “whole”), consists of instances of B (referred to as the “parts”).

Aggregation

The multiplicity of the relationship determines how aggregation relationships are implemented.

If an instance of class A contains one instance of class B, the relationship is implemented by storing the reference to an instance of B in A.

If an instance of class A contains more than one instance of B, some sort of data structure or collection will need to be used to store the object references of type B.

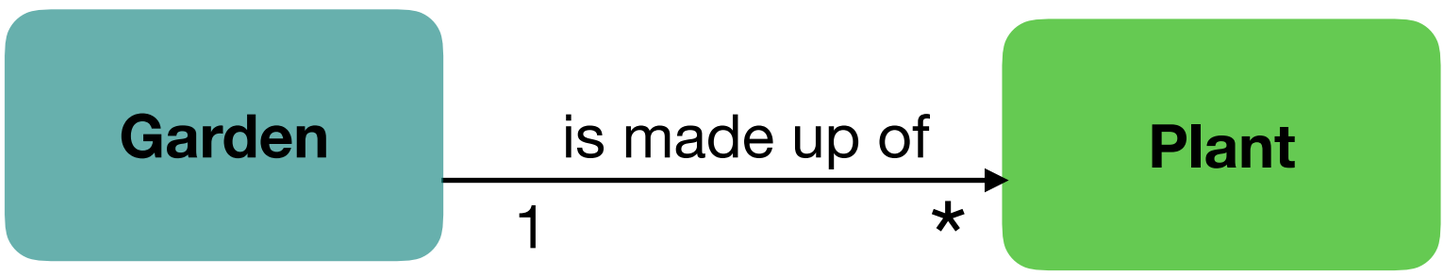
Example - Implementing Aggregation

```
public class Company{
    private Department[] department; //declaration
    public Company( ){
      department = new Department[5]; //initialisation
    }
}
```

Aggregation Relationship

Example - Implementing Aggregation

```
public class Garden{ //Whole
    private Plant[] plants; //Part
    public Garden( ){
      plants = new Plant[100];
    }
}
```



Aggregation Relationship

Aggregation

Aggregation raises the issue of ownership.

Our example for the abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted).

In other words, the lifetime of a garden and its plants are independent:

Reference: Page 62, Chapter 3 - G. Booch

Aggregation

Aggregation raises the issue of ownership.

Our example for the abstraction of a garden permits different plants to be raised in a garden over time, but replacing a plant does not change the identity of the garden as a whole, nor does removing a garden necessarily destroy all of its plants (they are likely just transplanted).

In other words, the lifetime of a garden and its plants are independent:

Example - Implementing Aggregation

```
public class Garden{
      private Plant[] plants;
      private int numPlants; // keeps track of number of
                                    // plants in the garden
      public Garden( ){
        plants = new Plant[100]; //100 plants permitted
        numPlants = 0; // Garden has 0 plants
      }
      public void addPlant(Plant freshPlant){
         if(numPlants < 100)
           plants[numPlants] = freshPlant;
      }
      public void removePlant(Plant p){
         for(int i = 0; i <numPlants; i++){ //go through the garden
             Plant gardenPlant = plants[i]; //locate the first plant
             if(gardenPlant.equals(p)){ // equality based on some criteria
                 // remove this plant and shift plant objects up in the array
                 }
         }
      }
}
```

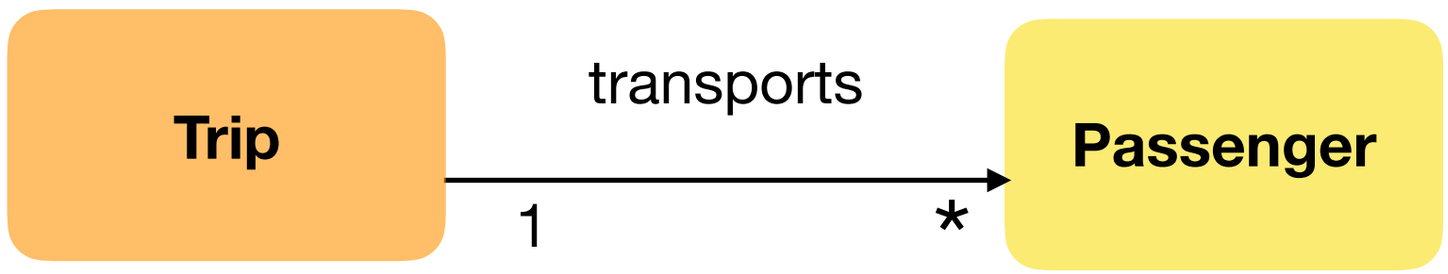
Composition

Composition is a form of aggregation with stronger ownership between the “whole” and its “parts”.

The “parts” in the relationship live and die with the “whole”. In other words, the “whole” is responsible for the creation and destruction of its “parts”.

Example - Implementing Composition

```
public class Trip{
    private Passenger[] passengers; //declaration
    private Date date;
    public Trip(Date tripDate){
      passengers = new Passenger[60]; //initialisation
      date = tripDate;
    }
}
```



Composition Relationship

A Trip doesn't need passengers to exist

A Passenger can only exist if associated with a trip