

Aggregation and Composition



EECS2030 F: Advanced
Object Oriented Programming
Fall 2022

CHEN-WEI WANG

Learning Outcomes

This module is designed to help you learn about:

- **Call by Value**: Primitive vs. Reference Argument Values
- Aggregation vs. Composition: **Terminology** and **Modelling**
- **Aggregation**: Building Sharing Links & **Navigating** Objects
- **Composition**: Implementation via **Copy Constructors**
- **Design Decision**: Aggregation or Composition?

Call by Value (1)

- Consider the general form of a call to some *mutator method* `m`, with *context object* `co` and **argument value** `arg`:

```
co.m(arg)
```

- Argument variable `arg` is *not* passed directly to the method call.
- Instead, argument variable `arg` is passed indirectly: a **copy** of the value stored in `arg` is made and passed to the method call.
- What can be the type of variable `arg`? [Primitive or Reference]
 - `arg` is primitive type (e.g., `int`, `char`, `boolean`, etc.):
Call by Value: Copy of `arg`'s **stored value** (e.g., `2`, `'j'`, `true`) is made and passed.
 - `arg` is reference type (e.g., `String`, `Point`, `Person`, etc.):
Call by Value: Copy of `arg`'s **stored reference/address** (e.g., `Point@5cb0d902`) is made and passed.

Call by Value (2.1)

For illustration, let's assume the following variant of the `Point` class:

```
public class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() { return this.x; }
    public int getY() { return this.y; }
    public void moveVertically(int y){ this.y += y; }
    public void moveHorizontally(int x){ this.x += x; }
}
```

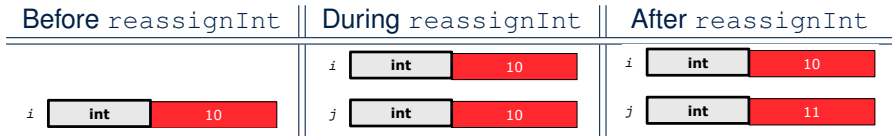
Call by Value (2.2.1)

```
public class Util {  
    void reassignInt(int j) {  
        j = j + 1; }  
    void reassignRef(Point q) {  
        Point np = new Point(6, 8);  
        q = np; }  
    void changeViaRef(Point q) {  
        q.moveHorizontally(3);  
        q.moveVertically(4); } }
```

```
1 @Test  
2 public void testCallByVal() {  
3     Util u = new Util();  
4     int i = 10;  
5     assertTrue(i == 10);  
6     u.reassignInt(i);  
7     assertTrue(i == 10);  
8 }
```

- **Before** the mutator call at **L6**, **primitive** variable `i` stores 10.
- **When** executing the mutator call at **L6**, due to **call by value**, a copy of variable `i` is made.
 - ⇒ The assignment `i = i + 1` is only effective on this copy, not the original variable `i` itself.
- ∴ **After** the mutator call at **L6**, variable `i` still stores 10.

Call by Value (2.2.2)



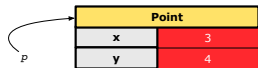
Call by Value (2.3.1)

<pre> public class Util { void reassignInt(int j) { j = j + 1; } void reassignRef(Point q) { Point np = new Point(6, 8); q = np; } void changeViaRef(Point q) { q.moveHorizontally(3); q.moveVertically(4); } } </pre>	1 2 3 4 5 6 7 8 9 10	<pre> @Test public void testCallByRef_1() { Util u = new Util(); Point p = new Point(3, 4); Point refOfPBefore = p; u.reassignRef(p); assertTrue(p == refOfPBefore); assertTrue(p.getX() == 3); assertTrue(p.getY() == 4); } </pre>
--	---	---

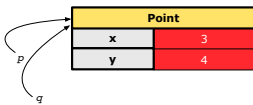
- **Before** the mutator call at **L6**, **reference** variable `p` stores the **address** of some `Point` object (whose `x` is 3 and `y` is 4).
- **When** executing the mutator call at **L6**, due to **call by value**, a **copy of address** stored in `p` is made.
 - ⇒ The assignment `p = np` is only effective on this copy, not the original variable `p` itself.
- ∴ **After** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`).

Call by Value (2.3.2)

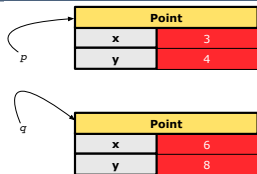
Before `reassignRef`



During `reassignRef`



After `reassignRef`



Call by Value (2.4.1)

```

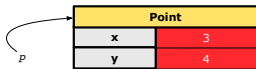
public class Util {
    void reassignInt(int j) {
        j = j + 1; }
    void reassignRef(Point q) {
        Point np = new Point(6, 8);
        q = np; }
    void changeViaRef(Point q) {
        q.moveHorizontally(3);
        q.moveVertically(4); } }
1
2
3
4
5
6
7
8
9
10
@Test
public void testCallByRef_2() {
    Util u = new Util();
    Point p = new Point(3, 4);
    Point refOfPBefore = p;
    u.changeViaRef(p);
    assertTrue(p == refOfPBefore);
    assertTrue(p.getX() == 6);
    assertTrue(p.getY() == 8);
}

```

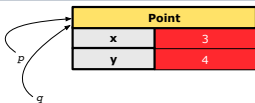
- **Before** the mutator call at **L6**, **reference** variable `p` stores the **address** of some `Point` object (whose `x` is 3 and `y` is 4).
- **When** executing the mutator call at **L6**, due to **call by value**, a **copy of address** stored in `p` is made. [**Alias**: `p` and `q` store same address.]
 ⇒ `q.moveHorizontally` impacts the **same object** referenced by `p` and `q`.
- ∴ **After** the mutator call at **L6**, variable `p` still stores the original address (i.e., same as `refOfPBefore`), but its `x` and `y` values have been modified via `q`.

Call by Value (2.4.2)

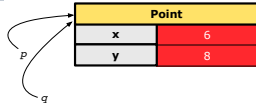
Before `changeViaRef`



During `changeViaRef`



After `changeViaRef`



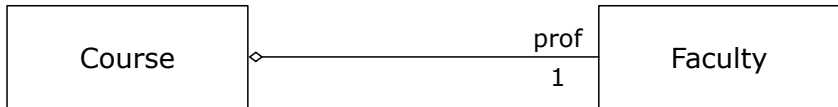
Aggregation vs. Composition: Terminology

Container object: an object that contains others.

Containee object: an object that is contained within another.

- e.g., Each course has a faculty member as its instructor.
 - **Container**: Course **Containee**: Faculty.
- e.g., Each student is registered in a list of courses; Each faculty member teaches a list of courses.
 - **Container**: Student, Faculty **Containees**: Course.
 e.g., eeecs2030 taken by jim (student) and taught by tom (faculty).
 ⇒ **Containees** may be **shared** by different instances of **containers**.
 e.g., When EECS2030 is finished, jim and jackie still exist!
 ⇒ **Containees** may **exist independently** without their **containers**.
- e.g., In a file system, each directory contains a list of files.
 - **Container**: Directory **Containees**: File.
 e.g., Each file has exactly one parent directory.
 ⇒ A **containee** may be **owned** by only one **container**.
 e.g., Deleting a directory also deletes the files it contains.
 ⇒ **Containees** may **co-exist** with their **containers**.

Aggregation: Independent Containees Shared by Containers (1.1)



```
public class Course {
    private String title;
    private Faculty prof;
    public Course(String title) {
        this.title = title;
    }
    public void setProf(Faculty prof) {
        this.prof = prof;
    }
    public Faculty getProf() {
        return this.prof;
    }
}
```

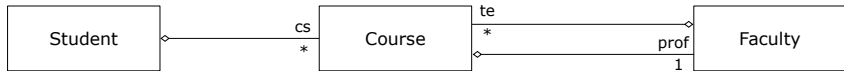
```
public class Faculty {
    private String name;
    public Faculty(String name) {
        this.name = name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
}
```

Aggregation: Independent Containees Shared by Containers (1.2)

```
@Test
public void testAggregation1() {
    Course eecs2030 = new Course("Advanced OOP");
    Course eecs3311 = new Course("Software Design");
    Faculty prof = new Faculty("Jackie");
    eecs2030.setProf(prof);
    eecs3311.setProf(prof);
    assertTrue(eecs2030.getProf() == eecs3311.getProf());
    /* aliasing */
    prof.setName("Jeff");
    assertTrue(eecs2030.getProf() == eecs3311.getProf());
    assertTrue(eecs2030.getProf().getName().equals("Jeff"));

    Faculty prof2 = new Faculty("Jonathan");
    eecs3311.setProf(prof2);
    assertTrue(eecs2030.getProf() != eecs3311.getProf());
    assertTrue(eecs2030.getProf().getName().equals("Jeff"));
    assertTrue(eecs3311.getProf().getName().equals("Jonathan"));
}
```

Aggregation: Independent Containees Shared by Containers (2.1)



```
public class Student {
    private String id; Course[] cs; int noc; /* # of courses */
    public Student(String id) { ... }
    public void addCourse(Course c) { ... }
    public Course[] getCS() { ... }
}
```

```
public class Course { private String title; private Faculty prof; }
```

```
public class Faculty {
    private String name; Course[] te; int not; /* # of teaching */
    public Faculty(String name) { ... }
    public void addTeaching(Course c) { ... }
    public Course[] getTE() { ... }
}
```

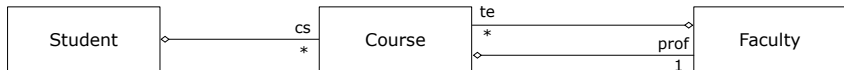
Aggregation: Independent Containees Shared by Containers (2.2)

```
@Test
public void testAggregation2() {
    Faculty p = new Faculty("Jackie");
    Student s = new Student("Jim");
    Course eecs2030 = new Course("Advanced OOP");
    Course eecs3311 = new Course("Software Design");
    eecs2030.setProf(p);
    eecs3311.setProf(p);
    p.addTeaching(eecs2030);
    p.addTeaching(eecs3311);
    s.addCourse(eecs2030);
    s.addCourse(eecs3311);

    assertTrue(eecs2030.getProf() == s.getCS()[0].getProf());
    assertTrue(s.getCS()[0].getProf()
        == s.getCS()[1].getProf());
    assertTrue(eecs3311 == s.getCS()[1]);
    assertTrue(s.getCS()[1] == p.getTE()[1]);
}
```

The Dot Notation (3.1)

In real life, the relationships among classes are sophisticated.



```

public class Student {
    private String id;
    private Course[] cs;
}
  
```

```

public class Course {
    private String title;
    private Faculty prof;
}
  
```

```

public class Faculty {
    private String name;
    private Course[] te;
}
  
```

- Assume: **private** attributes and **public** accessors
- **Aggregation links** between classes constrain how you can **navigate** among these classes.
- In the context of class `Student`:
 - Writing **`cs`** denotes the array of registered courses.
 - Writing **`cs[i]`** (where `i` is a valid index) navigates to the class `Course`, which changes the context to class `Course`.

OOP: The Dot Notation (3.2)

```
public class Student {  
    private String id;  
    private Course[] cs;  
}
```

```
public class Course {  
    private String title;  
    private Faculty prof;  
}
```

```
public class Faculty {  
    private String name;  
    private Course[] te;  
}
```

```
public class Student {  
    ... /* attributes */  
    /* Get the student's id */  
    public String getID() { return this.id; }  
    /* Get the title of the ith course */  
    public String getTitle(int i) {  
        return this.cs[i].getTitle();  
    }  
    /* Get the instructor's name of the ith course */  
    public String getName(int i) {  
        return this.cs[i].getProf.getName();  
    }  
}
```

OOP: The Dot Notation (3.3)

```
public class Student {  
    private String id;  
    private Course[] cs;  
}
```

```
public class Course {  
    private String title;  
    private Faculty prof;  
}
```

```
public class Faculty {  
    private String name;  
    private Course[] te;  
}
```

```
public class Course {  
    ... /* attributes */  
    /* Get the course's title */  
    public String getTitle() { return this.title; }  
    /* Get the instructor's name */  
    public String getName() {  
        return this.prof.getName();  
    }  
    /* Get title of ith teaching course of the instructor */  
    public String getTitle(int i) {  
        return this.prof.getTE()[i].getTitle();  
    }  
}
```

OOP: The Dot Notation (3.4)

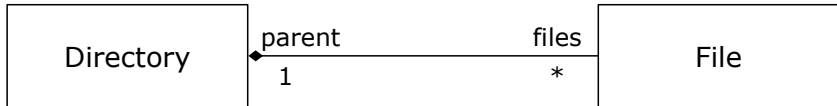
```
public class Student {  
    private String id;  
    private Course[] cs;  
}
```

```
public class Course {  
    private String title;  
    private Faculty prof;  
}
```

```
public class Faculty {  
    private String name;  
    private Course[] te;  
}
```

```
public class Faculty {  
    ... /* attributes */  
    /* Get the instructor's name */  
    public String getName() {  
        return this.name;  
    }  
    /* Get the title of ith teaching course */  
    public String getTitle(int i) {  
        return this.te[i].getTitle();  
    }  
}
```

Composition: Dependent Containees Owned by Containers (1.1)



Requirement: Files are not shared among directories.

Assume: **private** attributes and **public** accessors

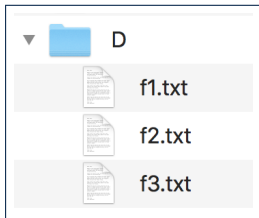
```

class File {
    String name;
    File(String name) {
        this.name = name;
    }
}
  
```

```

class Directory {
    String name;
    File[] files;
    int nof; /* num of files */
    Directory(String name) {
        this.name = name;
        files = new File[100];
    }
    void addFile(String fileName) {
        files[nof] = new File(fileName);
        nof++;
    }
}
  
```

Composition: Dependent Containees Owned by Containers (1.2.1)

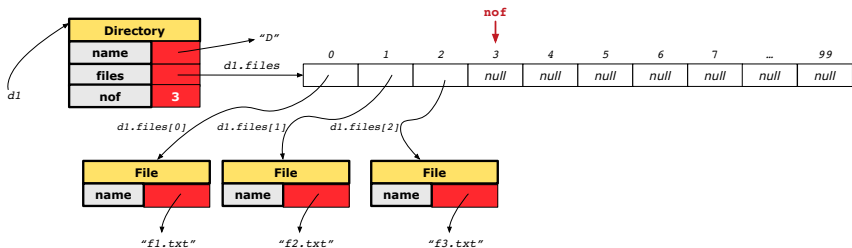


```
1 @Test
2 public void testComposition() {
3     Directory d1 = new Directory("D");
4     d1.addFile("f1.txt");
5     d1.addFile("f2.txt");
6     d1.addFile("f3.txt");
7     assertTrue(d1.getFiles()[0].getName().equals("f1.txt"));
8 }
```

- **L4:** 1st `File` object is created and **owned exclusively** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L5:** 2nd `File` object is created and **owned exclusively** by `d1`. No other directories are sharing this `File` object with `d1`.
- **L6:** 3rd `File` object is created and **owned exclusively** by `d1`. No other directories are sharing this `File` object with `d1`.

Composition: Dependent Containees Owned by Containers (1.2.2)

Right before test method `testComposition` terminates:



Composition: Dependent Containees Owned by Containers (1.3)

Problem: Implement a **copy constructor** for `Directory`.

A **copy constructor** is a constructor which initializes attributes from the argument object `other` (of the **same type** `Directory`).

```
class Directory {
    Directory(Directory other) {
        /* Initialize attributes via attributes of 'other'. */
    }
}
```

Hints:

- The implementation should be consistent with the effect of copying and pasting a directory.
- Separate copies of files are created.

Composition: Dependent Containees Owned by Containers (1.4.1)

Version 1: *Shallow Copy* by copying all attributes using =.

```
class Directory {
    Directory(Directory other) {
        /* value copying for primitive type */
        nof = other.nof;
        /* address copying for reference type */
        name = other.name; files = other.files; } }

```

Is a shallow copy satisfactory to support composition?
i.e., Does it still forbid sharing to occur?

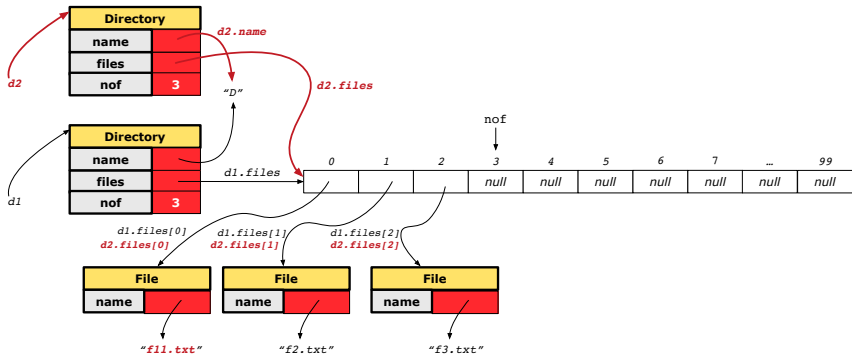
[**NO**]

```
@Test
public void testShallowCopyConstructor() {
    Directory d1 = new Directory("D");
    d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");
    Directory d2 = new Directory(d1);
    assertTrue(d1.getFiles() == d2.getFiles()); /* violation of composition */
    d2.getFiles()[0].changeName("f11.txt");
    assertFalse(d1.getFiles()[0].getName().equals("f1.txt"));
}

```


Composition: Dependent Containees Owned by Containers (1.4.2)

Right before test method `testShallowCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.5.1)

Version 2: a *Deep Copy*

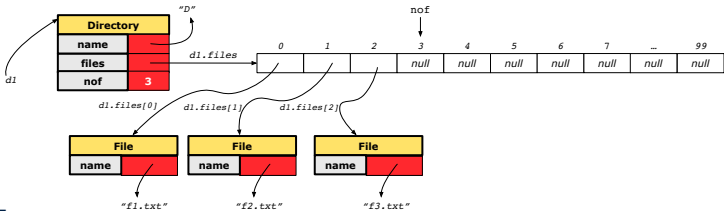
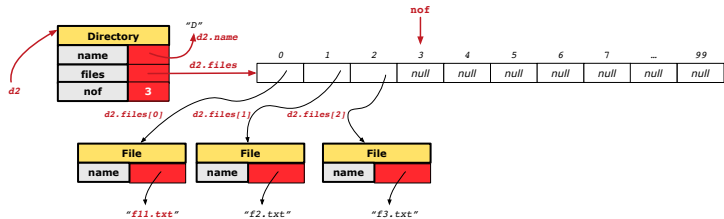
```
class File {  
    File(File other) {  
        this.name =  
            new String(other.name);  
    }  
}
```

```
class Directory {  
    Directory(String name) {  
        this.name = new String(name);  
        files = new File[100]; }  
    Directory(Directory other) {  
        this (other.name);  
        for(int i = 0; i < other.nof; i++) {  
            File src = other.files[i];  
            File nf = new File(src);  
            this.addFile(nf);  
        }  
    }  
    void addFile(File f) { ... }  
}
```

```
@Test  
public void testDeepCopyConstructor() {  
    Directory d1 = new Directory("D");  
    d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");  
    Directory d2 = new Directory(d1);  
    assertTrue(d1.getFiles() != d2.getFiles()); /* composition preserved */  
    d2.getFiles()[0].changeName("f11.txt");  
    assertTrue(d1.getFiles()[0].getName().equals("f1.txt"));  
}
```

Composition: Dependent Containees Owned by Containers (1.5.2)

Right before test method `testDeepCopyConstructor` terminates:



Composition: Dependent Containees Owned by Containers (1.5.3)

Q: Composition Violated?

```
class File {  
    File(File other) {  
        this.name =  
            new String(other.name);  
    }  
}
```

```
class Directory {  
    Directory(String name) {  
        this.name = new String(name);  
        files = new File[100]; }  
    Directory(Directory other) {  
        this (other.name);  
        for(int i = 0; i < other.nof; i++) {  
            File src = other.files[i];  
            this.addFile(src);  
        }  
    }  
    void addFile(File f) { ... }  
}
```

```
@Test  
public void testDeepCopyConstructor() {  
    Directory d1 = new Directory("D");  
    d1.addFile("f1.txt"); d1.addFile("f2.txt"); d1.addFile("f3.txt");  
    Directory d2 = new Directory(d1);  
    assertTrue(d1.getFiles() != d2.getFiles()); /* composition preserved */  
    d2.getFiles()[0].changeName("f11.txt");  
    assertTrue(d1.getFiles()[0] == d2.getFiles()[0]); /* composition violated! */  
}
```

Composition: Dependent Containees Owned by Containers (1.6)

Exercise: Implement the accessor in class `Directory`

```
class Directory {  
    File[] files;  
    int nof;  
    File[] getFiles() {  
        /* Your Task */  
    }  
}
```

so that it **preserves composition**, i.e., does not allow references of files to be shared.

Aggregation vs. Composition (1)

Terminology:

- **Container** object: an object that contains others.
- **Containee** object: an object that is contained within another.

Aggregation :

- Containees (e.g., Course) may be *shared* among containers (e.g., Student, Faculty).
- Containees *exist independently* without their containers.
- When a container is destroyed, its containees still exist.

Composition :

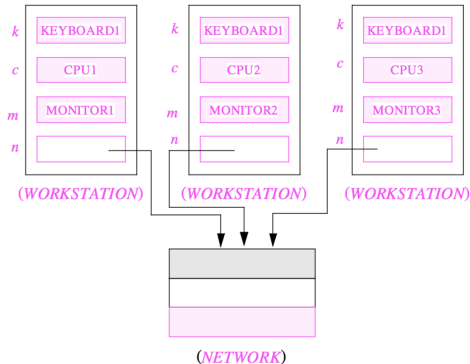
- Containers (e.g., Directory, Department) *own* exclusive access to their containees (e.g., File, Faculty).
- Containees cannot exist without their containers.
- Destroying a container destroys its containees *cascadingly*.

Aggregation vs. Composition (2)

Aggregations and *Compositions* may exist at the same time!

e.g., Consider a workstation:

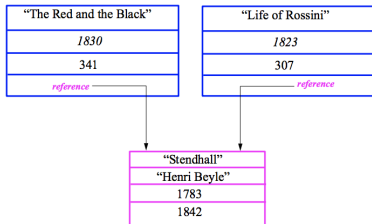
- Each workstation owns CPU, monitor, keyboard. [*compositions*]
- All workstations share the same network. [*aggregations*]



Aggregation vs. Composition (3)

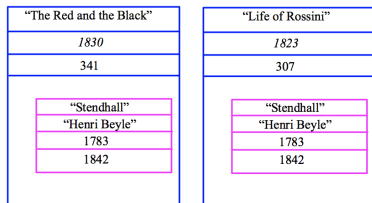
Problem: Every published book has an author. Every author may publish more than one books. Should the author field of a book be implemented as an *aggregation* or a *composition*?

author as an *aggregation*



Hyperlinked author page

author as a *composition*



Physical printed copies

Beyond this lecture...

Reproduce the *aggregation* and *composition* code examples in Eclipse.

Tip. Use the debugger to verify whether or not there is *sharing*.

Index (1)

Learning Outcomes

Call by Value (1)

Call by Value (2.1)

Call by Value (2.2.1)

Call by Value (2.2.2)

Call by Value (2.3.1)

Call by Value (2.3.2)

Call by Value (2.4.1)

Call by Value (2.4.2)

Aggregation vs. Composition: Terminology

Index (2)

**Aggregation: Independent Containees
Shared by Containers (1.1)**

**Aggregation: Independent Containees
Shared by Containers (1.2)**

**Aggregation: Independent Containees
Shared by Containers (2.1)**

**Aggregation: Independent Containees
Shared by Containers (2.2)**

The Dot Notation (3.1)

OOP: The Dot Notation (3.2)

OOP: The Dot Notation (3.3)

OOP: The Dot Notation (3.4)

Index (3)

**Composition: Dependent Containees
Owned by Containers (1.1)**

**Composition: Dependent Containees
Owned by Containers (1.2.1)**

**Composition: Dependent Containees
Owned by Containers (1.2.2)**

**Composition: Dependent Containees
Owned by Containers (1.3)**

**Composition: Dependent Containees
Owned by Containers (1.4.1)**

**Composition: Dependent Containees
Owned by Containers (1.4.2)**

Index (4)

**Composition: Dependent Containees
Owned by Containers (1.5.1)**

**Composition: Dependent Containees
Owned by Containers (1.5.2)**

**Composition: Dependent Containees
Owned by Containers (1.5.3)**

**Composition: Dependent Containees
Owned by Containers (1.6)**

Aggregation vs. Composition (1)

Aggregation vs. Composition (2)

Aggregation vs. Composition (3)

Beyond this lecture...