

Lecture 4

Part A

Call by Value - Primitive vs. Reference Arguments

Method Call: Callee vs. Caller

```
class A {  
    ...  
    void m(T param) {  
        /* use of param */  
    }  
}
```

header of method

parameter

declaration/definition
of a method
(callee)

```
class B {  
    ...  
    void n(..) {  
        A co = new A();  
        co.m(arg);  
    }  
}
```

Call by value
param = arg

argument
both cases of priv. & ref types

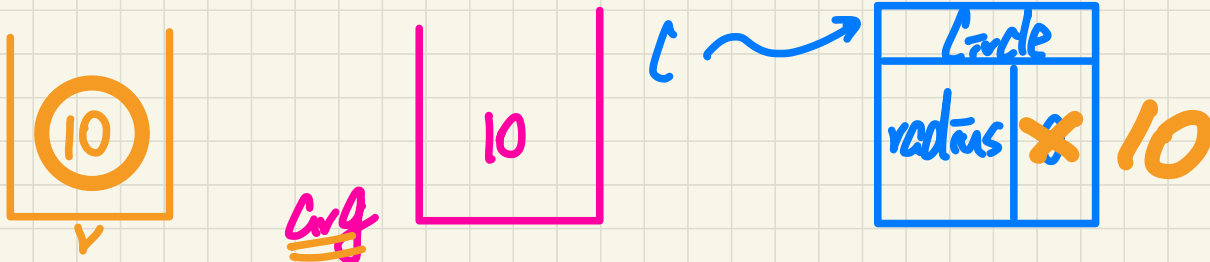
usage of a method
(caller)
capture context

context object

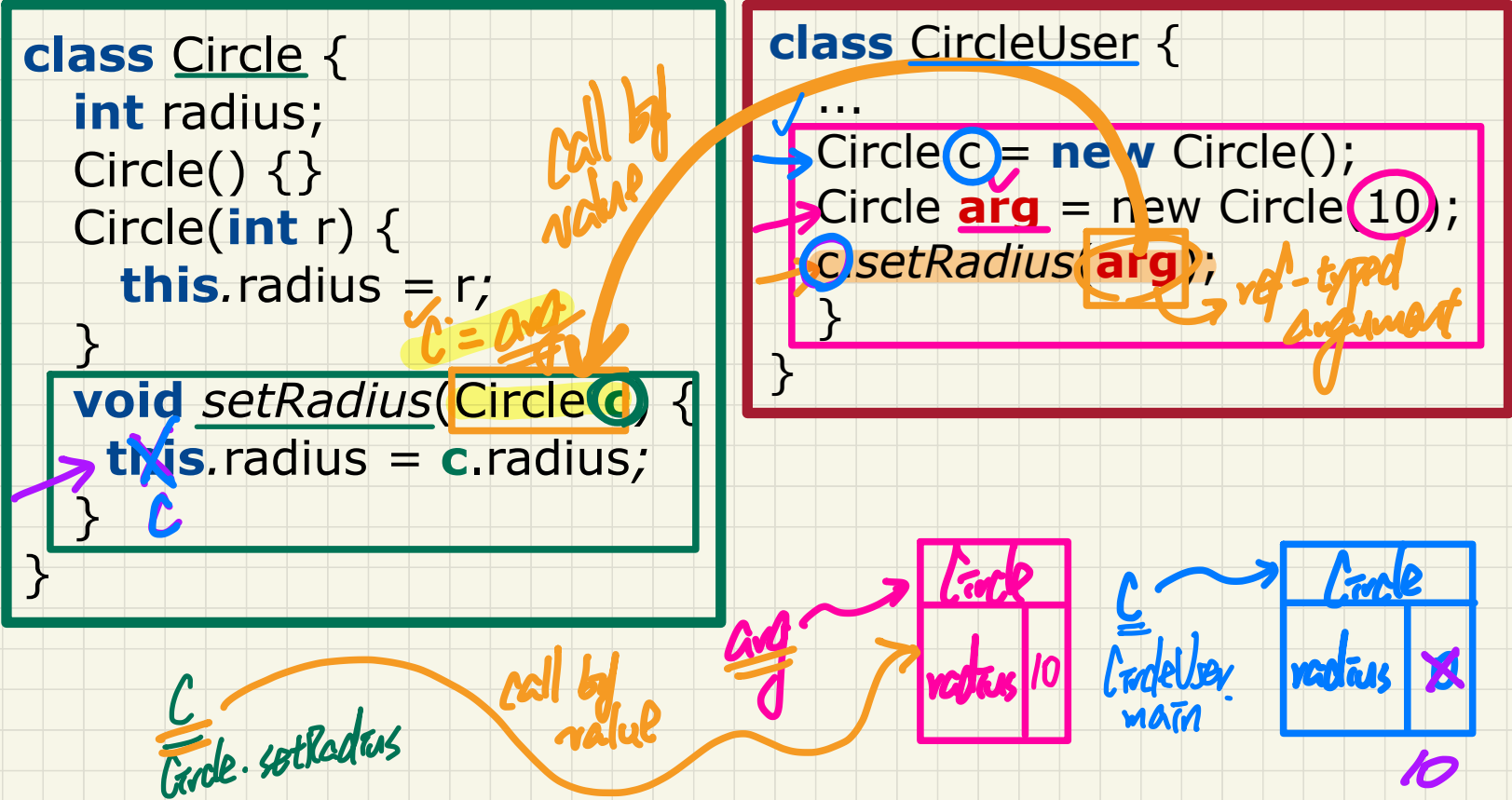
Call by Value: Primitive Argument

```
class Circle {  
    int radius; caller/def  
    void setRadius(int r) { r = arg  
        this.radius = r; param  
    }  
}
```

```
class CircleUser {  
    ... caller/application  
    Circle c = new Circle();  
    int arg = 10;  
    c.setRadius(arg); argument  
}
```



Call by Value: Reference Argument



Lecture 4

Part B

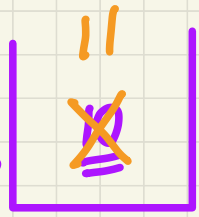
Call by Value - Asserting Call by Value in JUnit

Call by Value: Re-Assigning Primitive Parameter

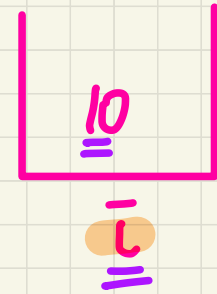
```
public class Util {  
    void reassignInt(int j) {  
        j = j + 1;  
    }  
    void reassignRef(Point q) {  
        Point np = new Point(6, 8);  
        q = np;  
    }  
    void changeViaRef(Point q) {  
        q.moveHorizontally(3);  
        q.moveVertically(4);  
    }  
}
```

```
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 }
```

Given that a copy of argument i is stored in j , when we execute $j = j + 1$, no change will be done to orig. i .



call by value
 $j = i$



argument
rather than i .

Call by Value: Re-Assigning Reference Parameter

Will executing this line redirect arg. p?

```
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);
    }
}
```

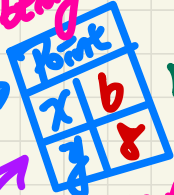
```
1 @Test
2 public void testCallByRef_1() {
3     Util u = new Util();
4     Point p = new Point(3, 4);
5     Point refOfPBefore = p;
6     u.reassignRef(p);
7     assertTrue(p == refOfPBefore);
8     assertTrue(p.getX() == 3);
9     assertTrue(p.getY() == 4);
10 }
```

call by value

q = P

p has not been re-assigned by the method.

only var being re-assigned



refOfPBefore

(orig) P

(parent) q



```
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: Calling Mutator on Reference Parameter

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

```
1 @Test
2 public void testCallByRef_2() {
3     Util u = new Util();
4     Point p = new Point(3, 4);
5     Point refOfPBefore = p;
6     u.changeViaRef(p);
7     assertTrue(p == refOfPBefore);
8     assertTrue(p.getX() == 6);
9     assertTrue(p.getY() == 8);
10 }
```

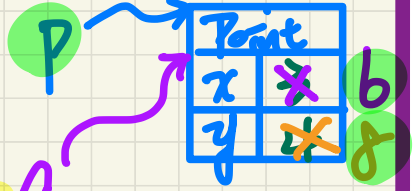
$q = p$
 param arg.

call by value:

caller side is able to

alias to (p)
 the arg. object
 is used as the
 context object →
 resulting in the
 object being
 altered. $q = p$

refOfPBefore



```
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; }
}
```

observe the
 change
 made via
 the param.
 alias (q)

call by value
 $y=4$ $x=3$

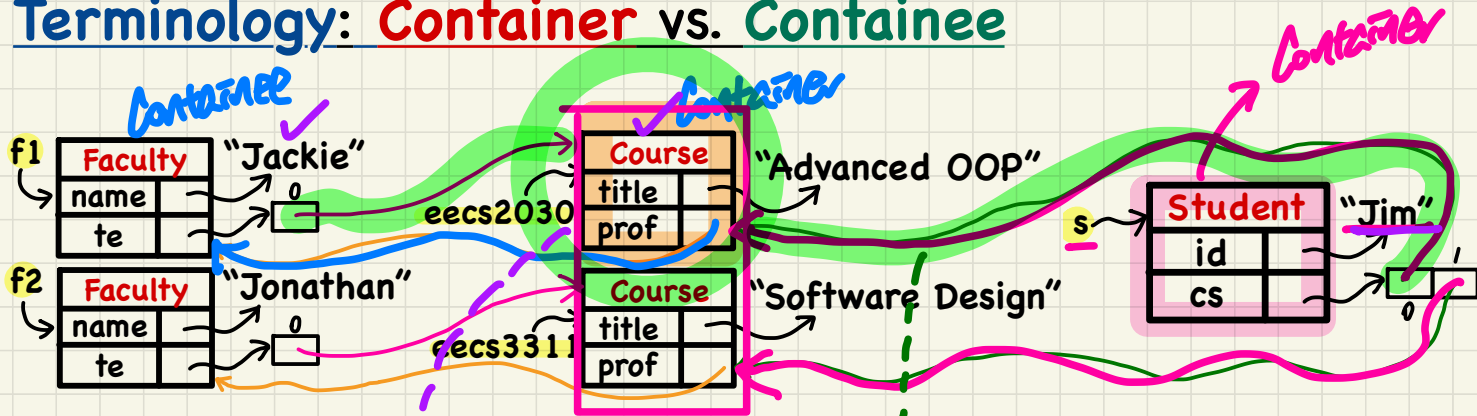
$q = p$
 $y=4$
 $x=3$

Lecture 4

Part C

Aggregation and Composition - Terminology, Modelling, and Implementation

Terminology: **Container** vs. **Containee**



When the **Container** (w.r.t. Faculty) or the **Container** (w.r.t. Student) is destroyed, **the other end should still exist.**

A container may be shared/contained by multiple containers.

Aggregation: Design

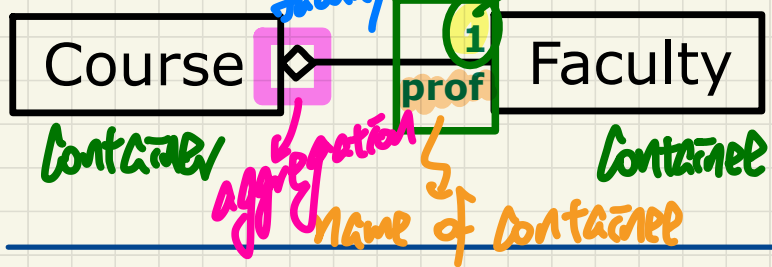
language independent

Java Implementation

language-specific

Design 1: Single Containee

A course contains a faculty as its prof. # of containees



```

class Course {
  Faculty prof;
  ...
}
  
```

Labels in code: Container (Course), single-valued containee (Faculty prof), name of containee (prof)

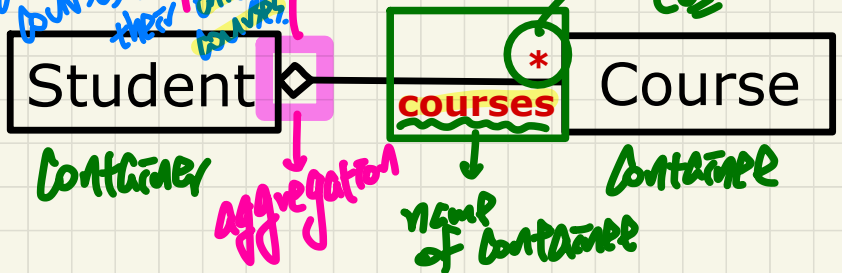
```

class Faculty {
  ...
}
  
```

Label in code: Containee (Faculty)

Design 2: Multiple Containees

A student contains a list of courses as their enrolled courses. multiple (2 or more)



```

class Student {
  Course[] courses;
  ...
}
  
```

Labels in code: multi-valued containees (Course[] courses), name of containee (courses)

```

class Course {
  ...
}
  
```

Label in code: Containee (Course)

Lecture 4

Part D

Aggregation and Composition - Building Aggregated Object Structure

Aggregation (1)

Course	
title	
prof	

Faculty	
name	

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

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

```
@Test
public void testAggregation() {
    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"));
}
```

Faculty	
name	

Course	
title	
prof	

Course	
title	
prof	

Faculty	
name	

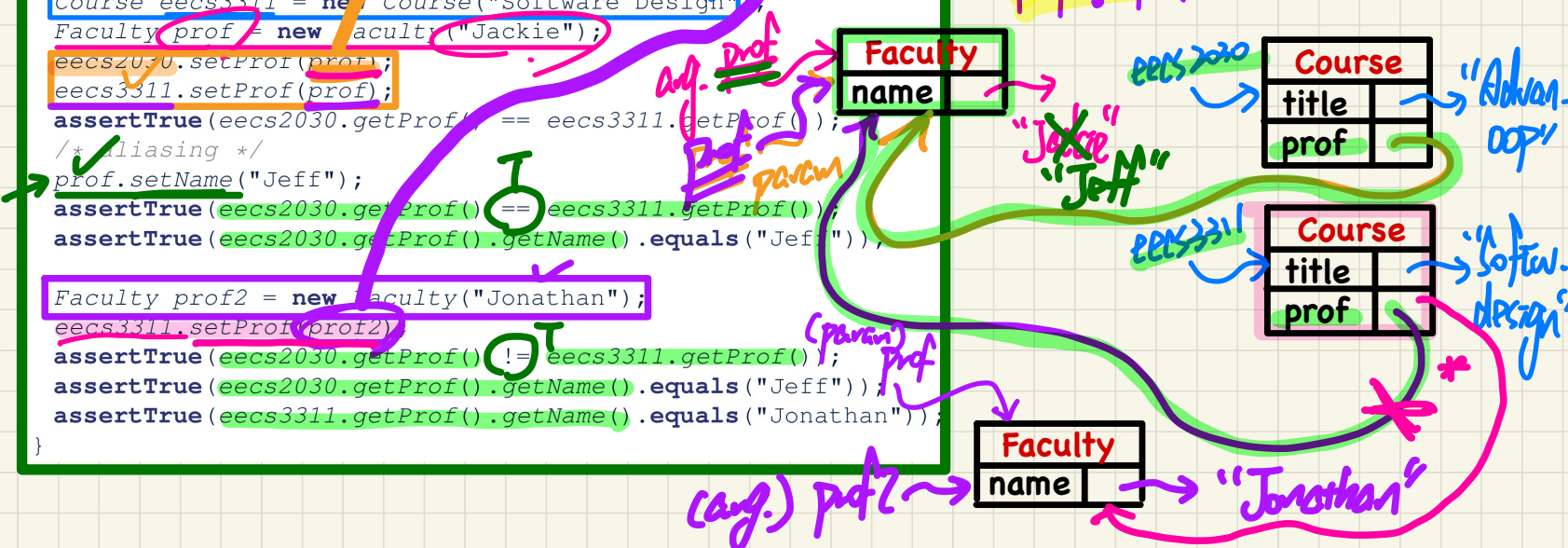
call by value: * eecs3311.prof = prof

prof = prof

call by value: prof = prof2

prof = prof

call by value: prof = prof2



Aggregation (2)

Student	
id	
cs	

Faculty	
name	
te	

Course	
title	
prof	

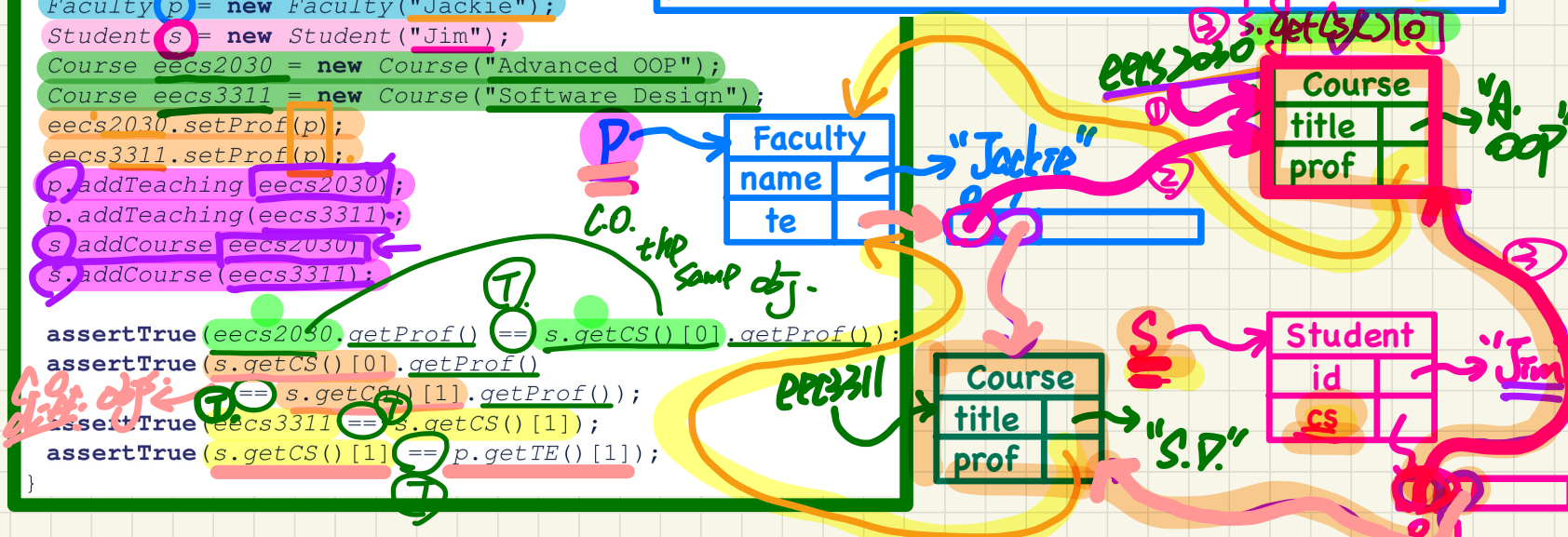
```
public class Student {
    private String id; Course[] cs; int noc; /* # of courses */
    public Student(String id) { ... }
    public void addCourse(Course c) { ... }
    public Course[] getCS() { ... }
}
```

```
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() { ... }
}
```

```
@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]);
}
```

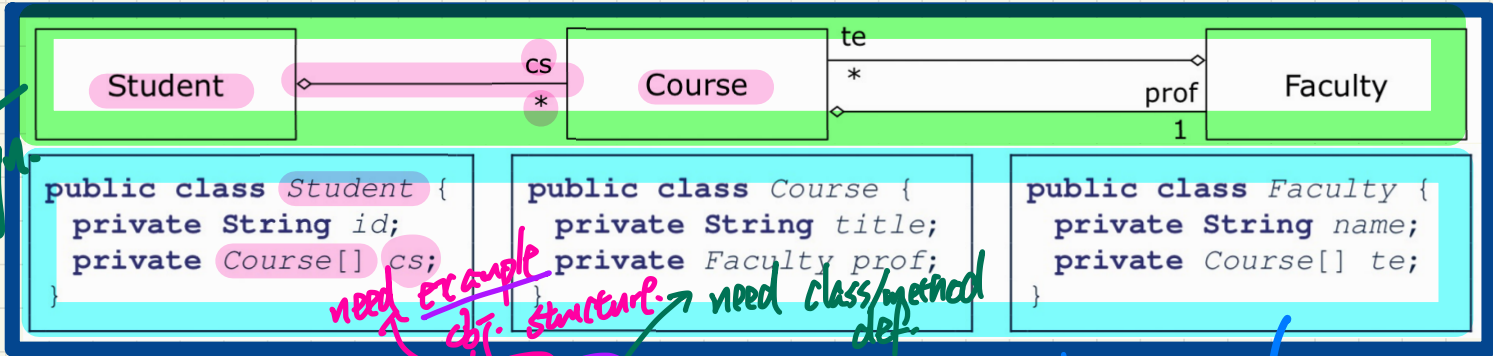


Lecture 4

Part E

Aggregation and Composition - Navigating Objects via Aggregation Links

Runtime Object Structure: Student, Course, Faculty



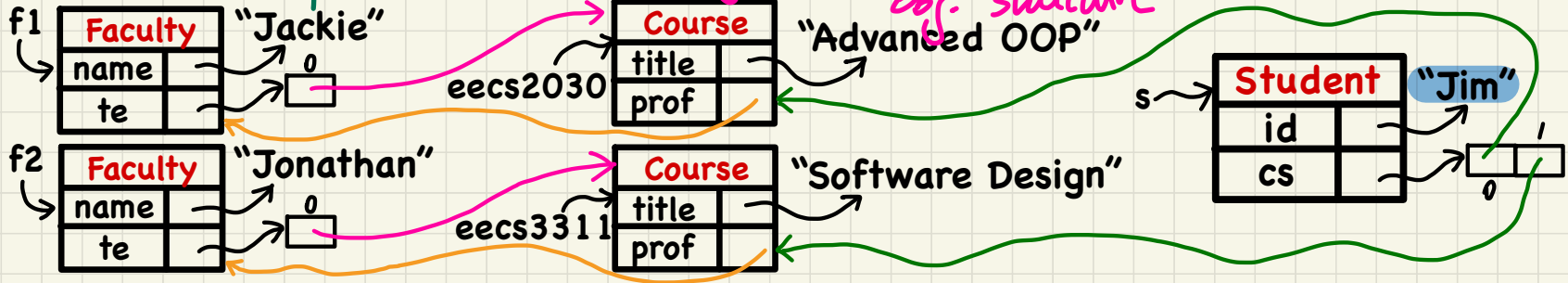
design

need example obj. structure. need class/method def.

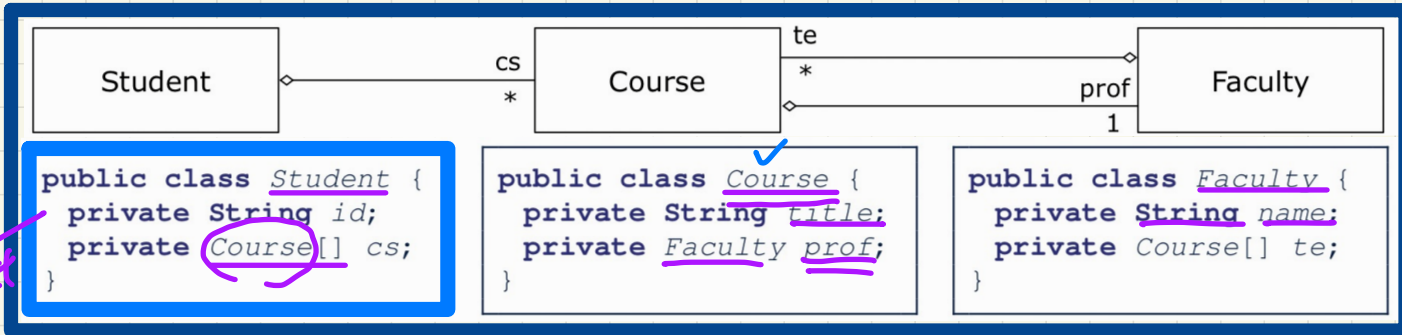
Knowing the context class

- ① using the type info. of attributes
- ② using the links in an example obj. structure

implement method implementation



Dot Notation for Navigating Classes (1)



```

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;
}
  
```

/* Get the student's id.

```

*/
String getID() {
  
```

return this.id;
Student String

/* Title of ith course

```

*/
String getTitle(int i) {
  
```

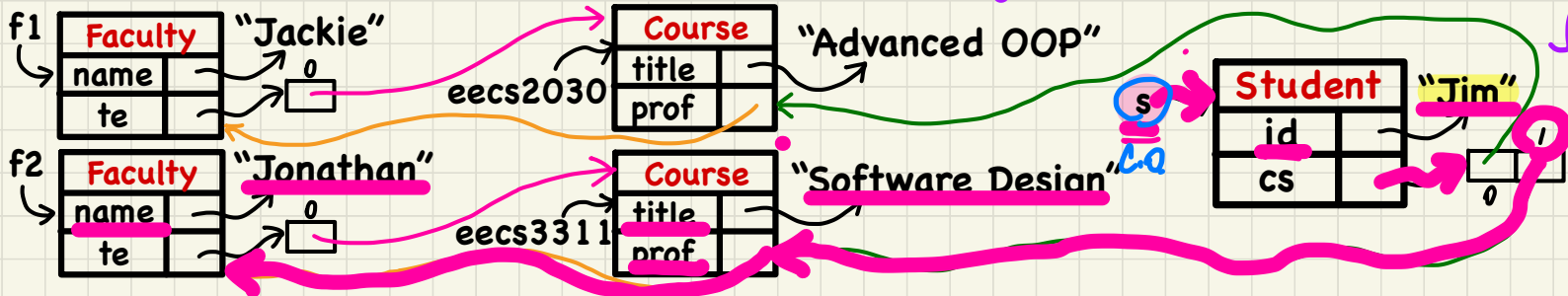
return this.cs[i].getTitle();
Student Course[] String

/* Name of

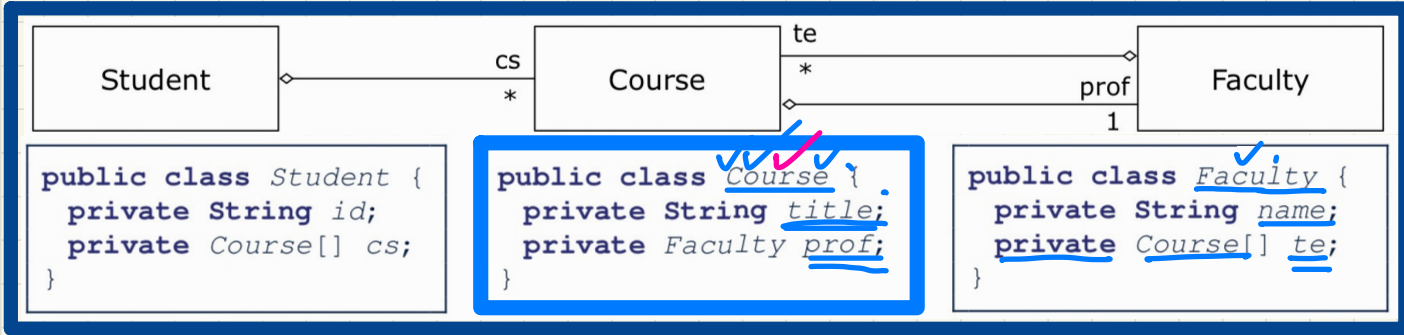
```

*/
String getName(int i) {
  
```

return this.cs[i].getProf().getName();
S. Course[] Faculty String



Dot Notation for Navigating Classes (2)



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

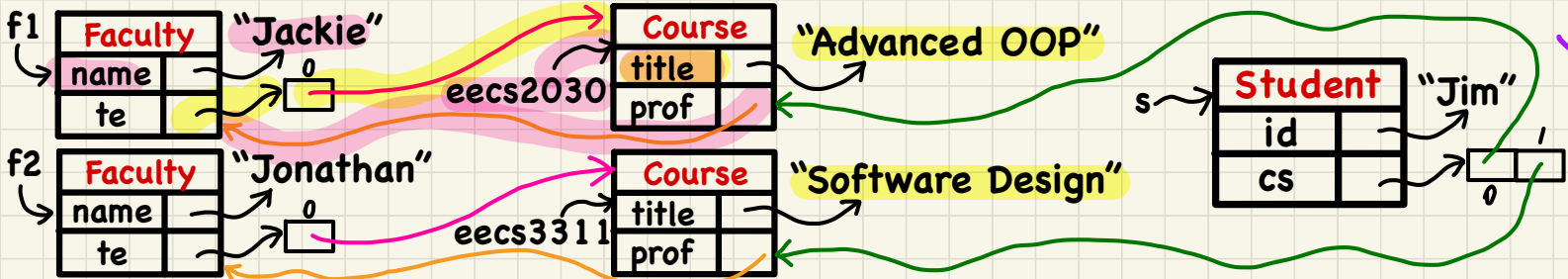
```
public class Course {
    private String title;
    private Faculty prof;
}
```

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

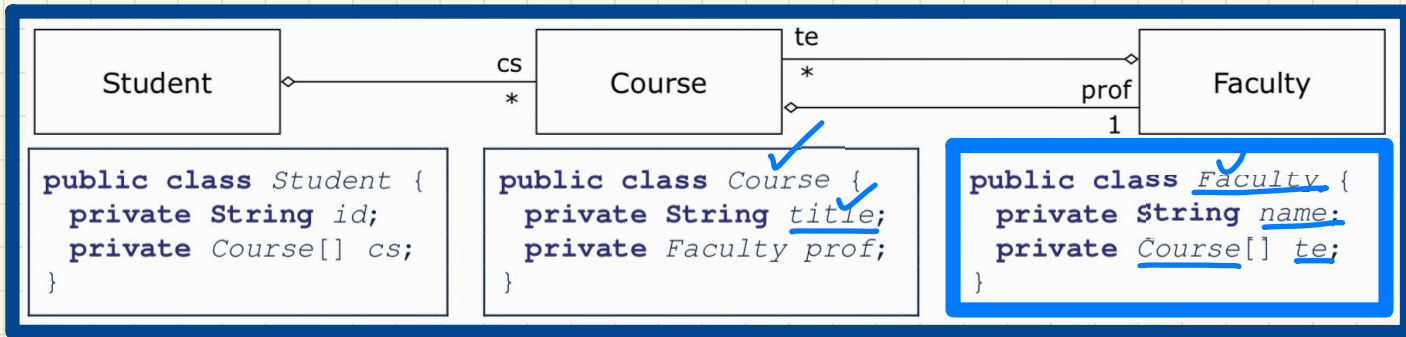
```
/* Get course's title.
 */
String getTitle() {
    return this.title;
}
```

```
/* Name of instructor
 */
String getName() {
    return this.prof.getName();
}
```

```
/* Title of instructor's
 * ith teaching course
 */
String getTitle(int i) {
    return this.prof.getTeachingCourse(i).getTitle();
}
```

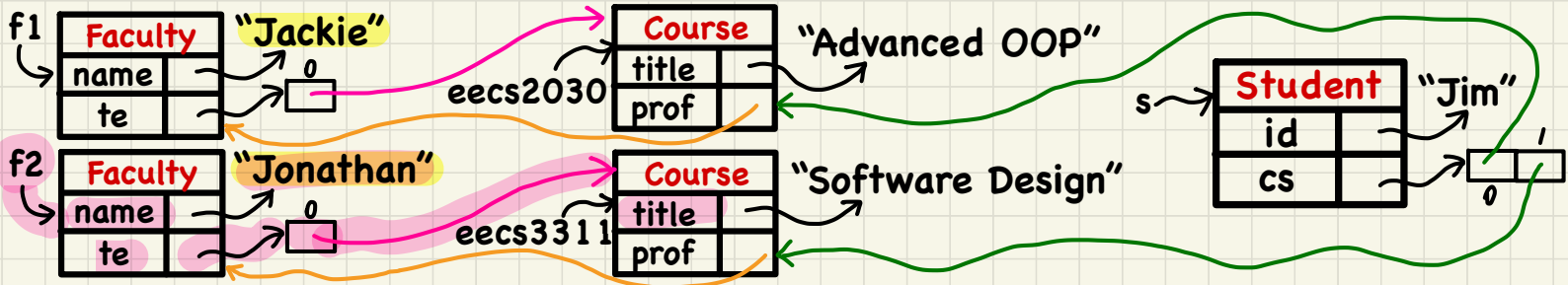


Dot Notation for Navigating Classes (3)



```
/* Name of instructor  
*/  
String getName() {  
    return this.name;  
}
```

```
/* Title of instructor's  
* ith teaching course  
*/  
String getTitle(int i) {  
    return this.te[i].getTitle();  
}
```



Lecture 4

Part F

***Aggregation and Composition -
Implementing
Composition via Copy Constructors***

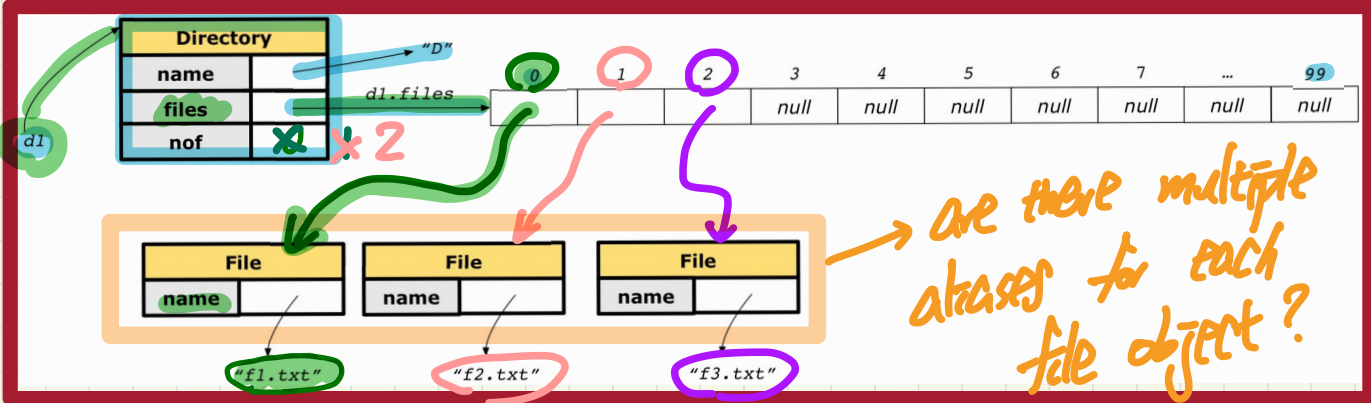
Composition: No Sharing

```
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 ++;  
    }  
}
```

Copy the ref of param name

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

```
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(  
8         d1.files[0].name.equals("f1.txt")  
9     )  
}
```



are there multiple aliases for each file object?

So far, it's a composition.
No ⇒ no sharing of files.

Composition: Copy Constructor (Shallow Copy)

```
@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.files == d2.files); /* violation of composition */
    d2.files[0].changeName("f11.txt");
    assertFalse(d1.files[0].name.equals("f1.txt"));
}
```

call by value: $d1 = d1$

aliases, allow sharing.
only looking addresses without creating new objects

```
class Directory {
    String name;
    File[] files;
    int nof; /* num of files */
    Directory(Directory other) {
        /* value copying for primitive type */
        nof = other.nof;
        /* address copying for reference type */
        files = other.files;
    }
}
```

(same other) (arg.)

no longer the case is modified via d2 → the same files array shared by d1 and d2.

Directory	
name	"D"
files	[f1.txt, f2.txt, f3.txt]
nof	3

Directory	
name	"D"
files	[f1.txt, f2.txt, f3.txt]
nof	3

d1.files	0	1	2	3	4	5	6	7	...	99
	f1.txt			null	null	null	null	null	null	null

File	
name	"f1.txt"

File	
name	"f2.txt"

File	
name	"f3.txt"

"f1.txt" "f2.txt" "f3.txt"

Composition: Copy Constructor (Deep Copy)

```
@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.files != d2.files);
    d2.files[0].changeName("f11.txt");
    assertTrue(d1.files[0].name.equals("f1.txt"));
}
```

```
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.files.length; i++) {
            File src = other.files[i];
            File nf = new File(src);
            this.addFile(nf);
        }
    }
    void addFile(File f) { ... }
}
```

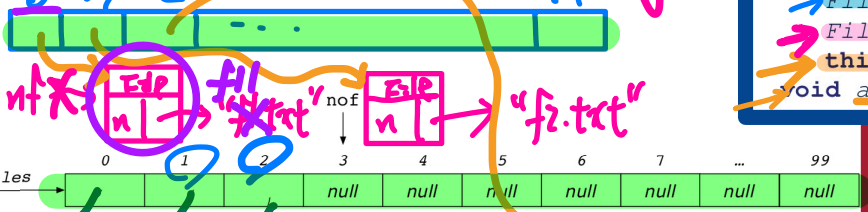
calling another overloaded constructor as a helper method.

not a copy constructor

unchanged ∴ deep copy

Directory	
name	"D"
files	[...]
nof	

Directory	
name	
files	[...]
nof	3



File	
name	"f1.txt"

File	
name	"f2.txt"

File	
name	"f3.txt"

- 0 other.files[0]
- 1 [1]
- 2 [2]

d2 other

src

nf

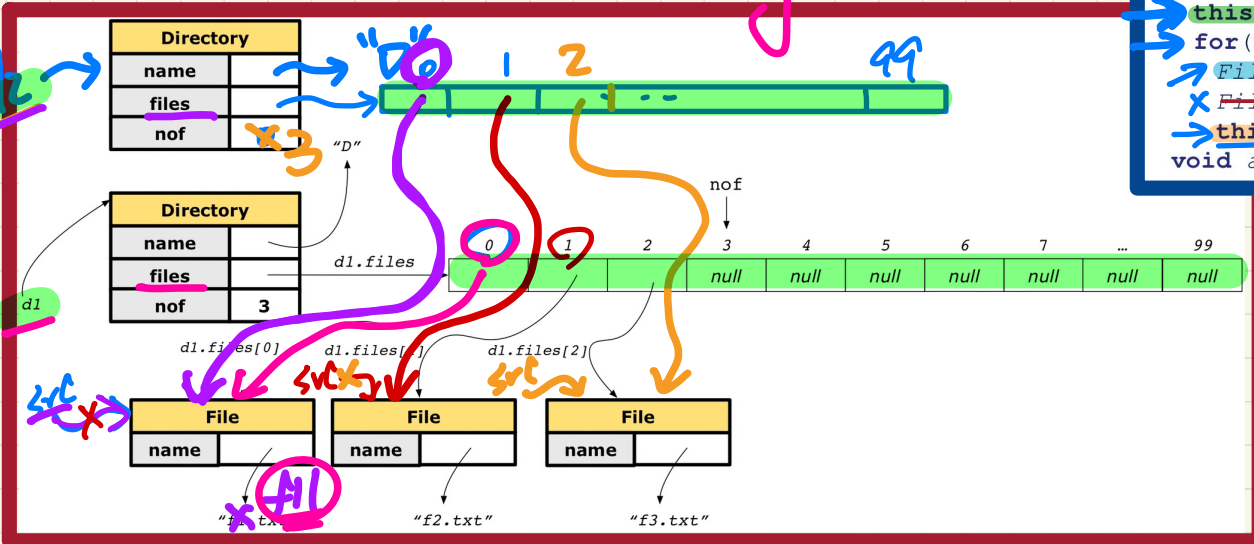
Exercise: Copy Constructor (Composition?)

```
@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.files != d2.files); /* composition preserved */
    d2.files[0].changeName("f11.txt");
    assertTrue(d1.files[0] == d2.files[0]); /* 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); other.
        for(int i = 0; i < nof; i++) {
            File src = other.files[i];
            File nf = new File(src);
            this.addFile(src);
        }
        void addFile(File f) { ... }
    }
}
```

↳ True if sharing.



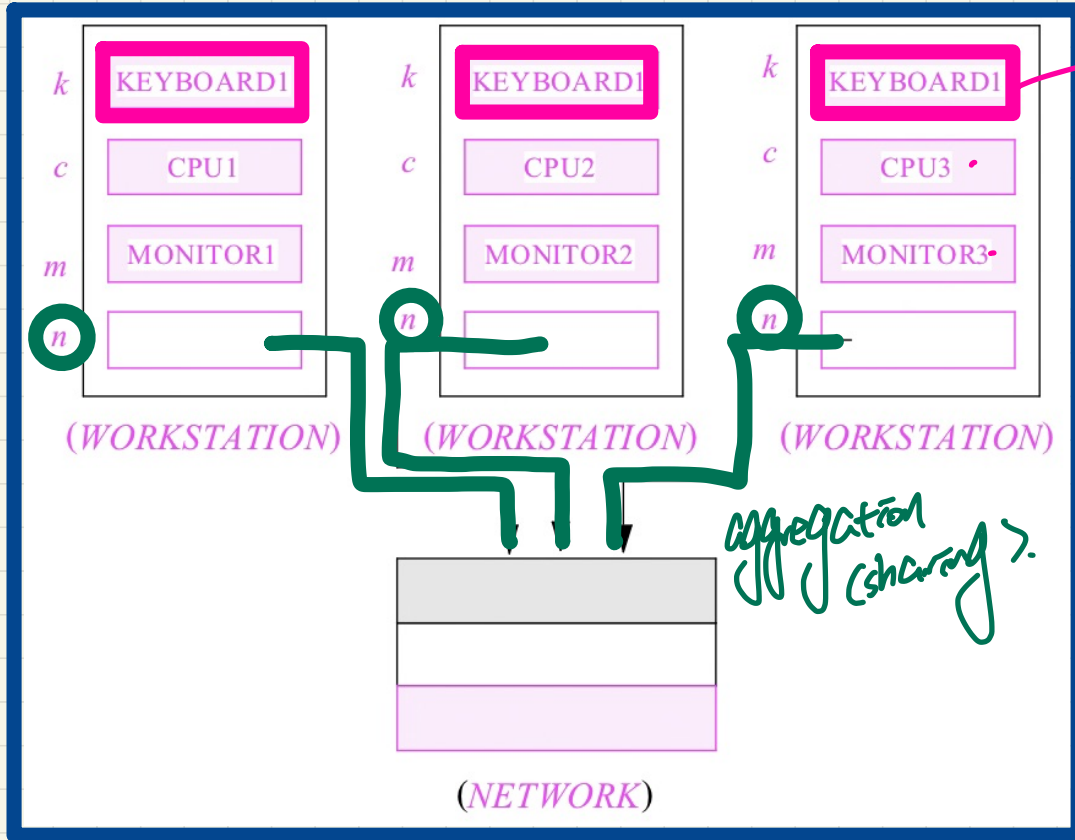
$\underline{0} = \underline{d2.files[0]} = \underline{0} \underline{2}$

Lecture 4

Part G

Aggregation and Composition - Example and Exercise

Modelling: Aggregation vs. Composition

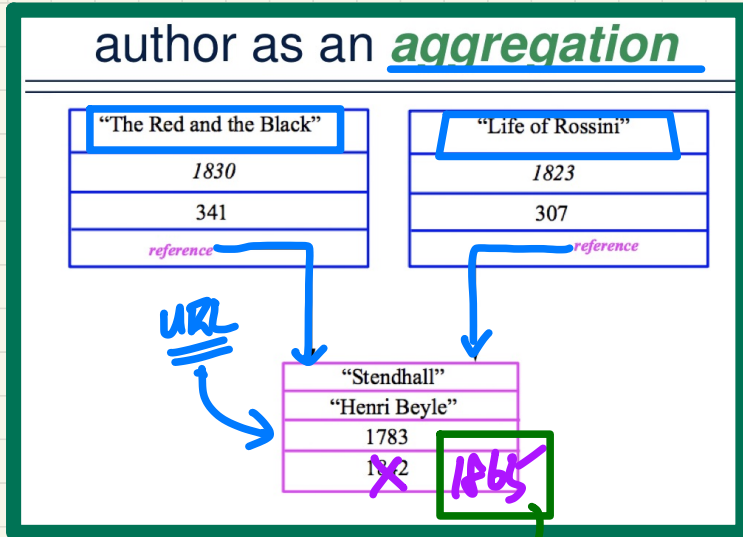


Exercise

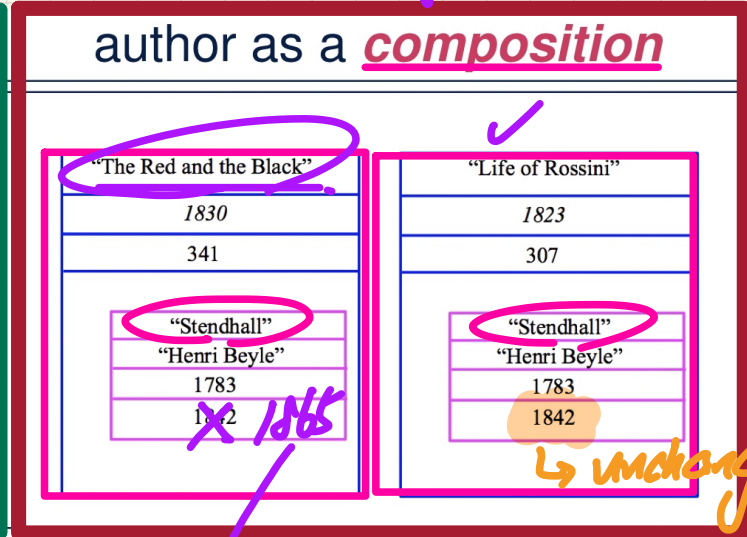
↳ declare Java classes and attributes for implementing this design.

Implementation: Aggregation or Composition

What if the author field gets modified?



Hyperlinked author page



Physical printed copies

change is visible to all containers (books) sharing the author page

change only applied to the owning container.