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 */
  }
}
```

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

header of method

parameter

declaration/definition of a method (callee)

Call by value

param = arg

argument

both cases of prim. & ref types.

context object

(application context) usage of a method (caller)

Call by Value: Primitive Argument

```
class Circle {
  int radius;
  void setRadius(int r) {
    this.radius = r;
  }
}
```

callee/def

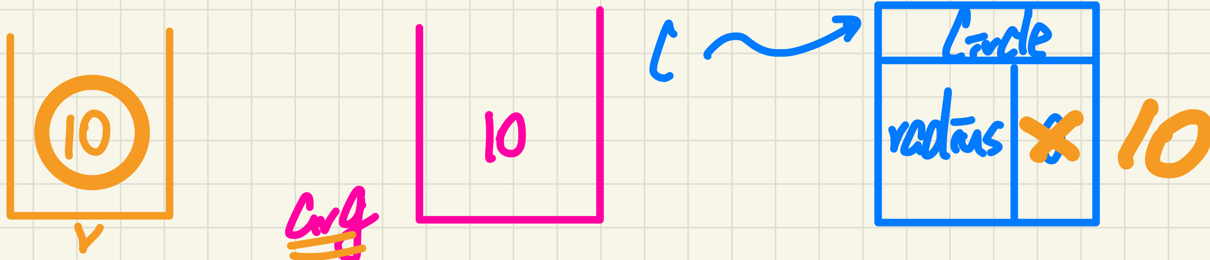
r = arg

param

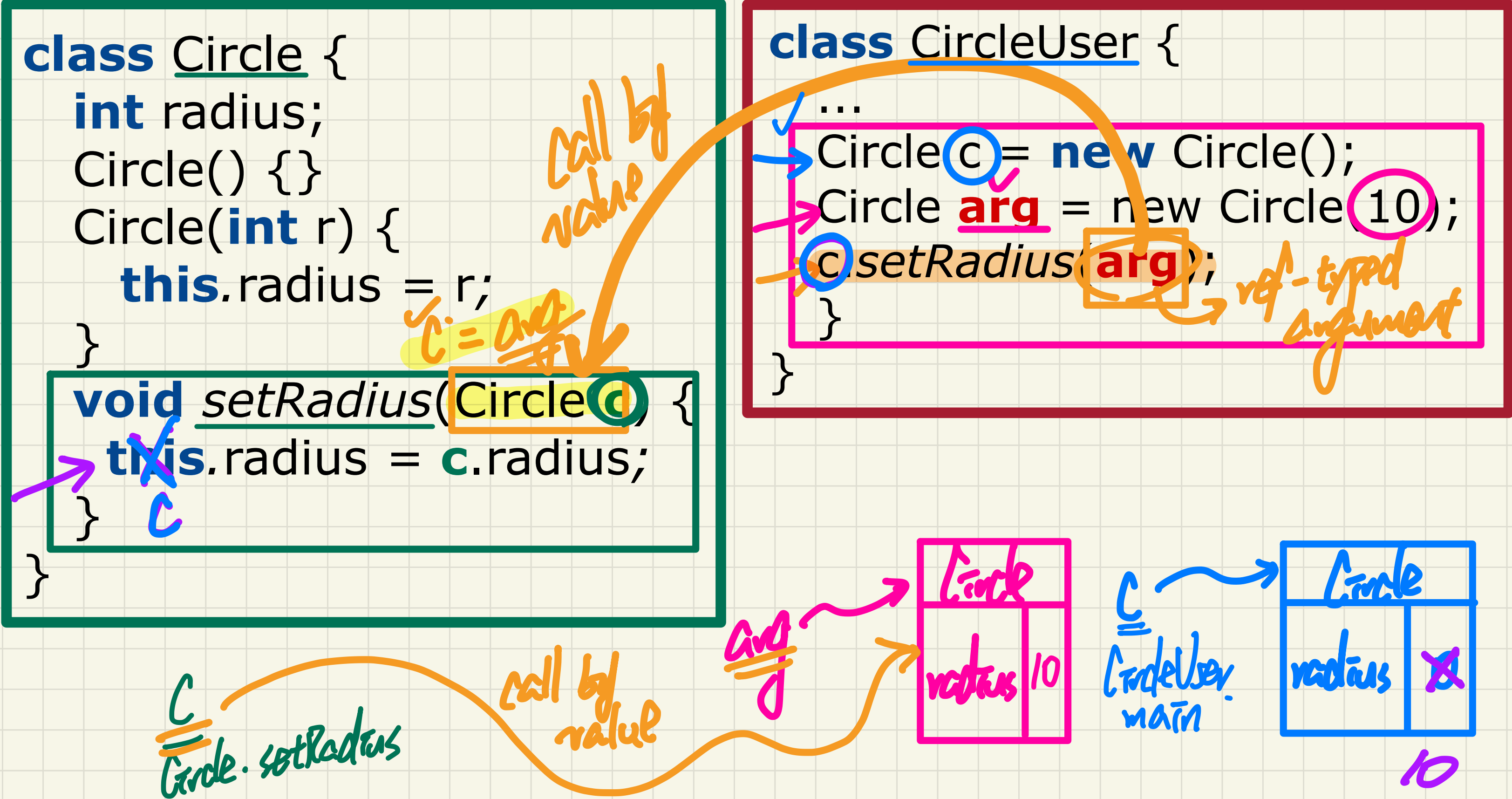
```
class CircleUser {
  ...
  Circle c = new Circle();
  int arg = 10;
  c.setRadius(arg);
  }
}
```

caller/application

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

```java
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); } }
```

```java
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  }
```

parameter

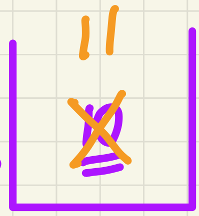
argument

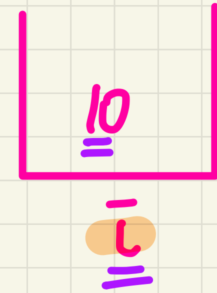
↳ rather than 11.

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

call by value

j = i

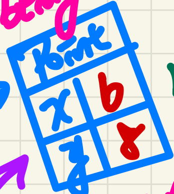


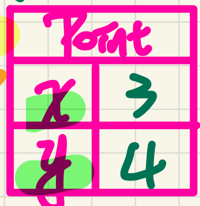


Call by Value: Re-Assigning Reference Parameter

```java
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); } }
```

```java
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 }
```





```java
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

```java
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); } }
```

```java
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 }
```

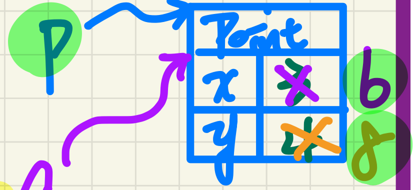
```java
public class Point {
  private int x;
  private int y;
  public Point(int x, int y) {
    this.x = x;
    this.y = y;
  }
  public int getX() { return this.x; }
  public int getY() { return this.y; }
  public void moveVertically(int y) { this.y += y; }
  public void moveHorizontally(int x) {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

call by value: q = p (param, arg.)

caller side is able to observe the change made via the param. alias (q)

alias to (q) the arg. object

q is used as the context object, resulting in the object being altered.



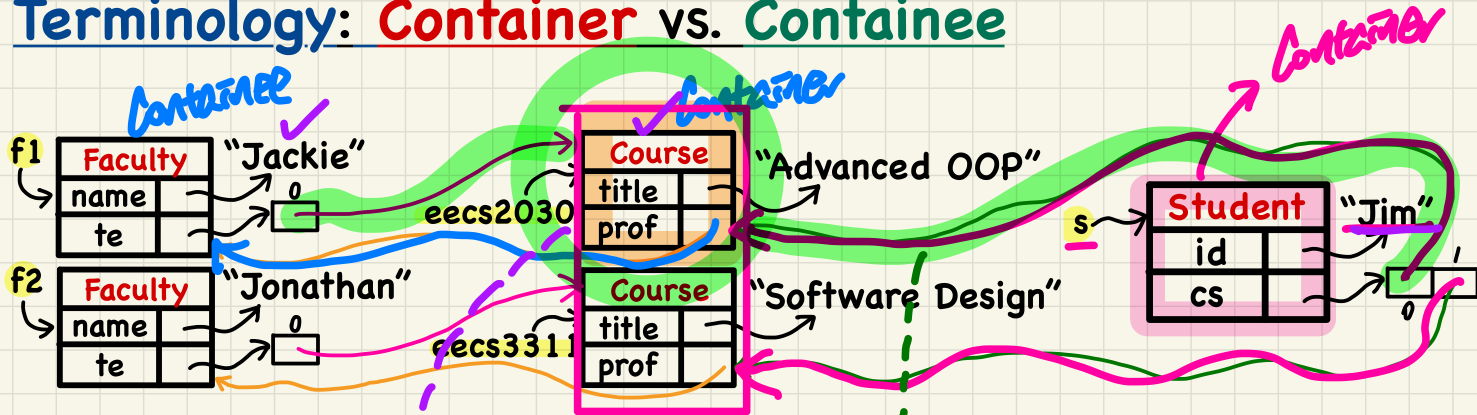
call by value: x = 3, y = 4

Lecture 4

Part C

Aggregation and Composition -
Terminology, Modelling, and Implementation

Terminology: Container vs. Containee

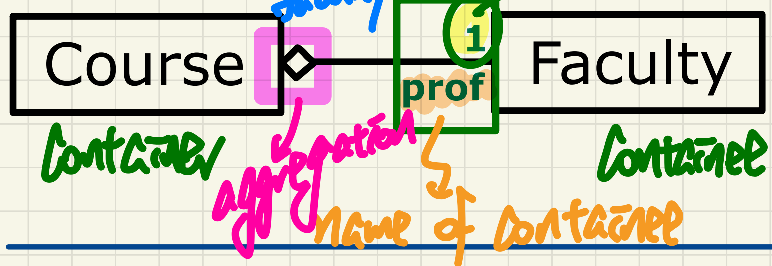


when the container (w.r.t. Faculty) or the containee (w.r.t. Student) is destroyed, the other end should still exist.

a containee may be shared/contained by multiple containers.

Aggregation: Design

Design 1: Single Containee

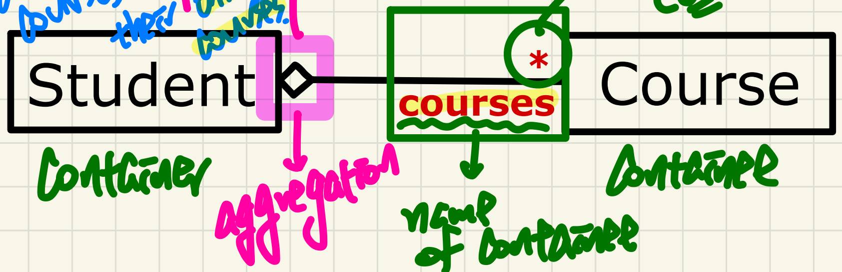


Java Implementation

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

```java
class Faculty {
  ...
}
```

Design 2: Multiple Containees




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

```java
class Course {
  ...
}
```

Lecture 4

Part D

***Aggregation and Composition -
Building Aggregated Object Structure***

Aggregation (1)

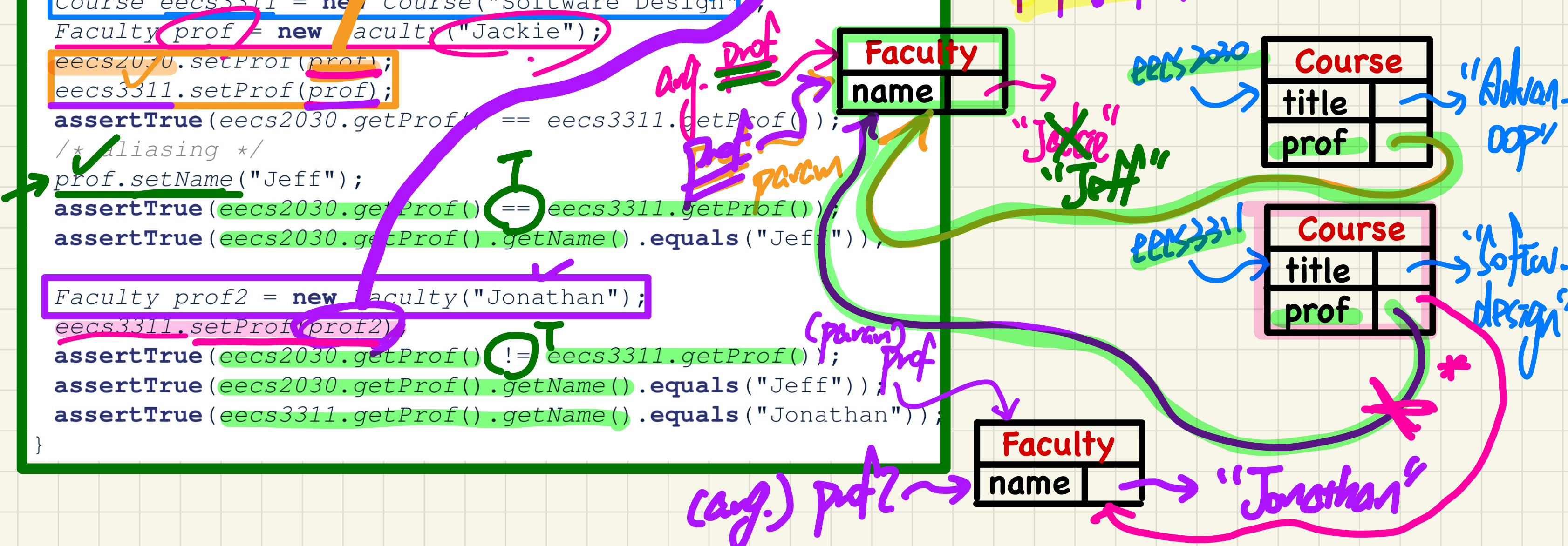
Course	
title	
prof	

Faculty	
name	

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

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

```java
@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 (2)

Student	
id	
cs	

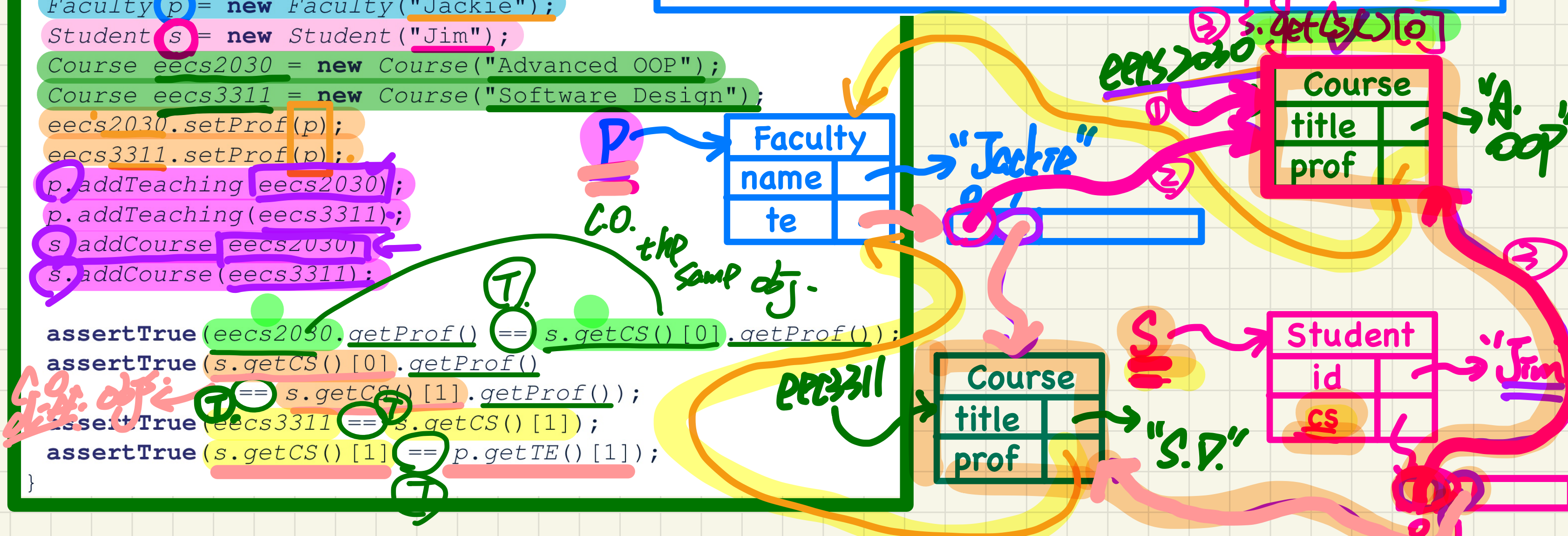
Faculty	
name	
te	

Course	
title	
prof	

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

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

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



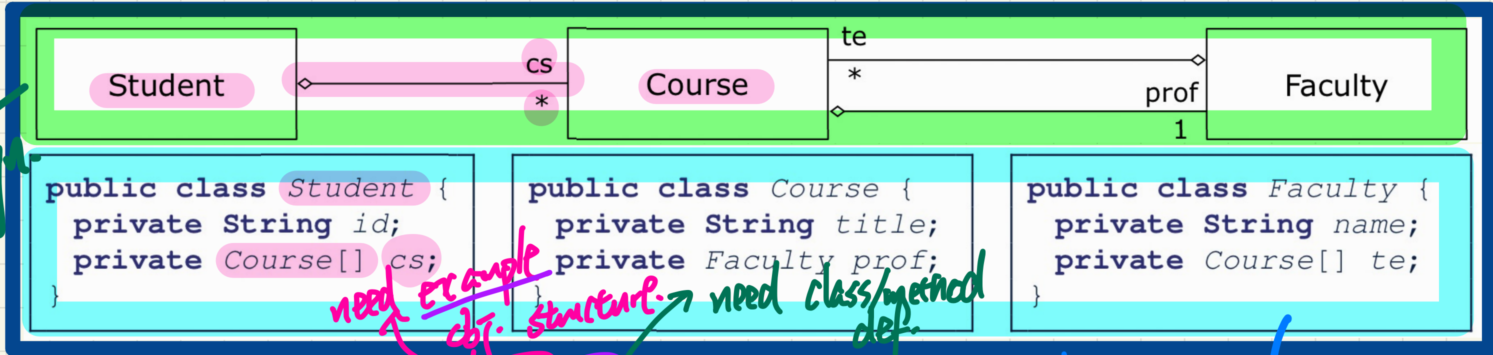
```java
@Test
public void testAggregation2() {
  Faculty p = new Faculty("Jackie");
  Student s = new Student("Jim");
  Course eecs2030 = new Course("Advanced OOP");
  Course eecs3311 = new Course("Software Design");
  eecs2030.setProf(p);
  eecs3311.setProf(p);
  p.addTeaching(eecs2030);
  p.addTeaching(eecs3311);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

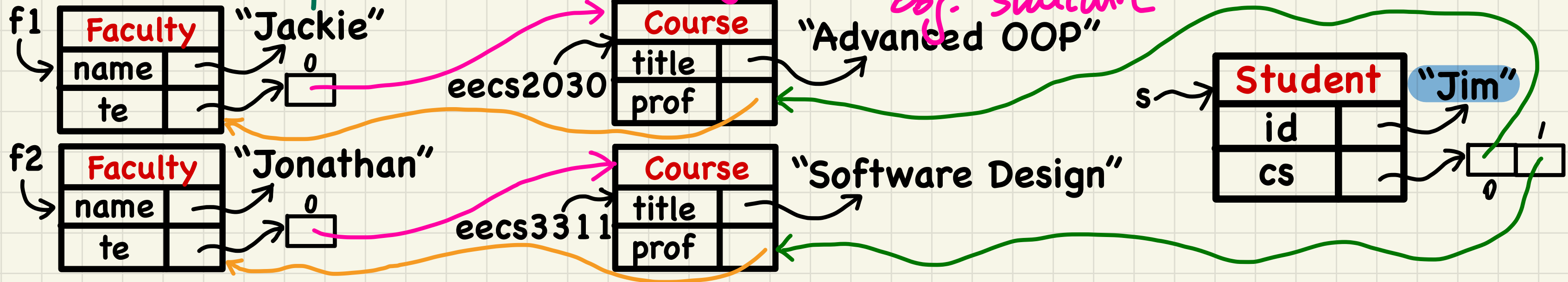
Lecture 4

Part E

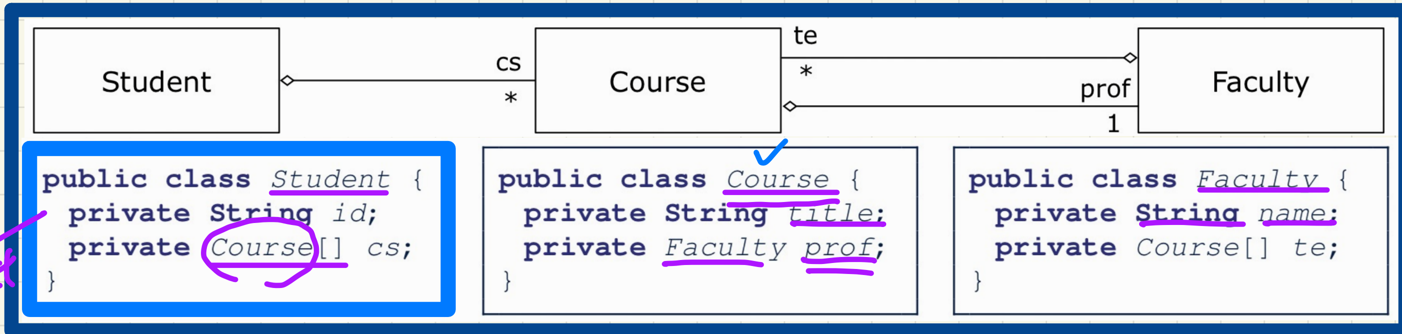
Aggregation and Composition -
Navigating Objects via Aggregation Links

Runtime Object Structure: Student, Course, Faculty





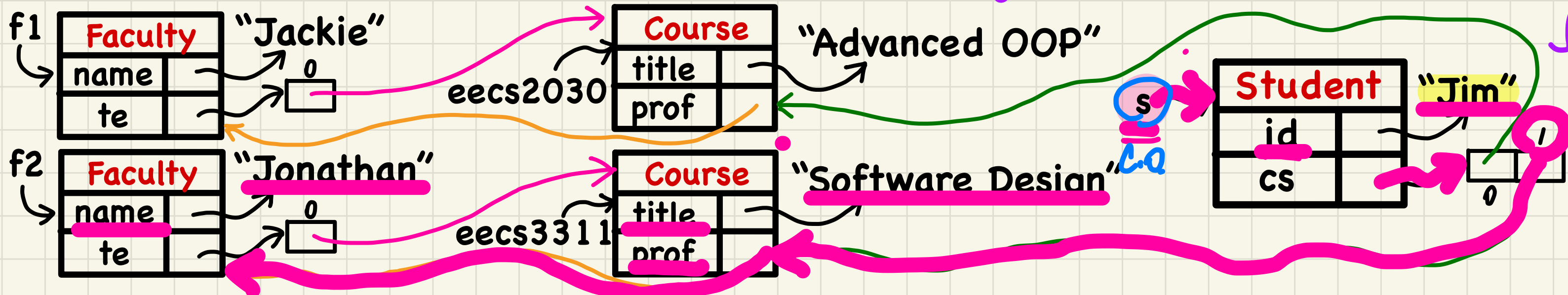
Dot Notation for Navigating Classes (1)



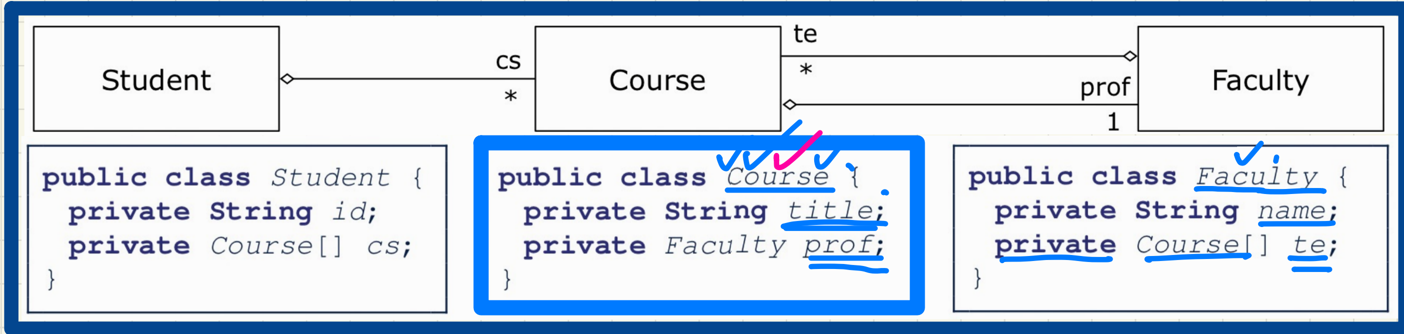

```
/* Get the student's id. */
String getID() {
  return this.id;
}
```

```
/* Title of ith course */
String getTitle(int i) {
  return this.cs[i].getTitle();
}
```

```
/* Name of
 * ith course's instructor
 */
String getName(int i) {
  return this.cs[i].getProf().getName();
}
```



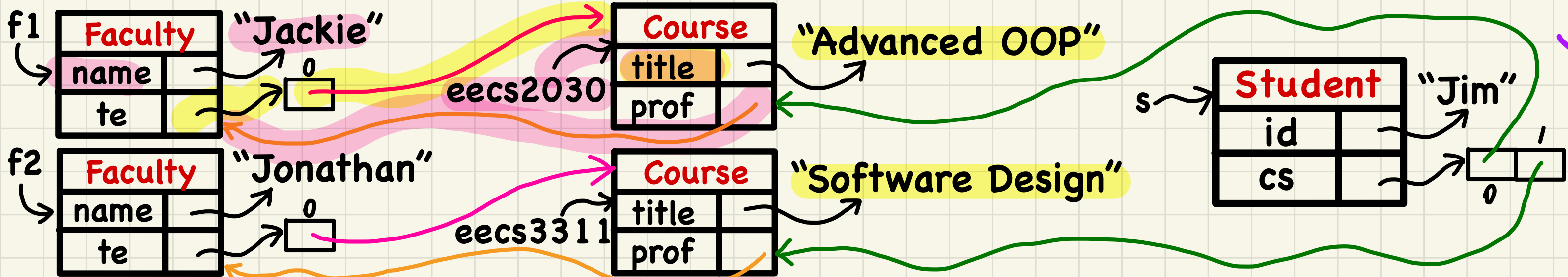
Dot Notation for Navigating Classes (2)



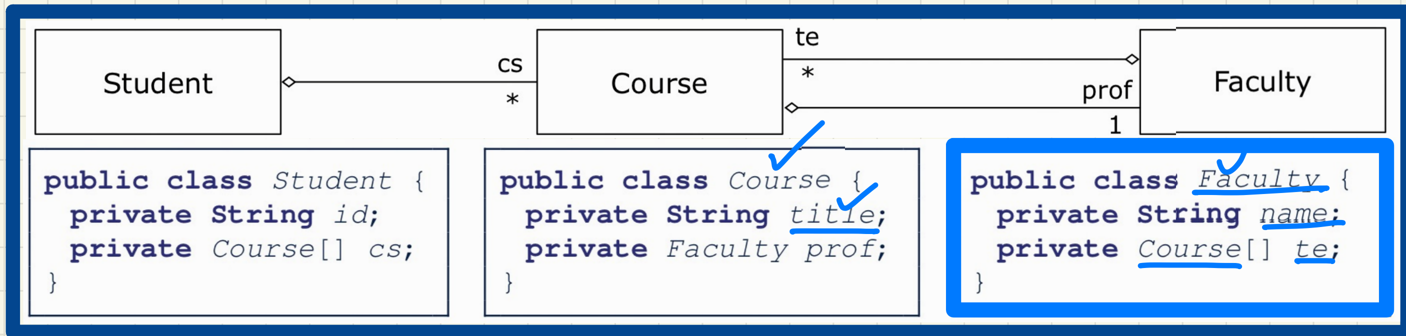

```
/* Get course's title.
 */
String getTitle() {
  return this.title;
}
```

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

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

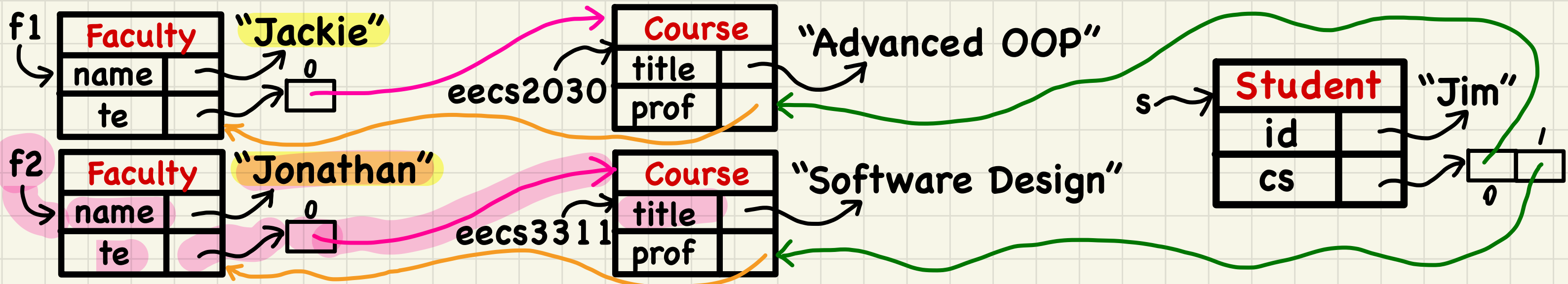


Dot Notation for Navigating Classes (3)




```
/* 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

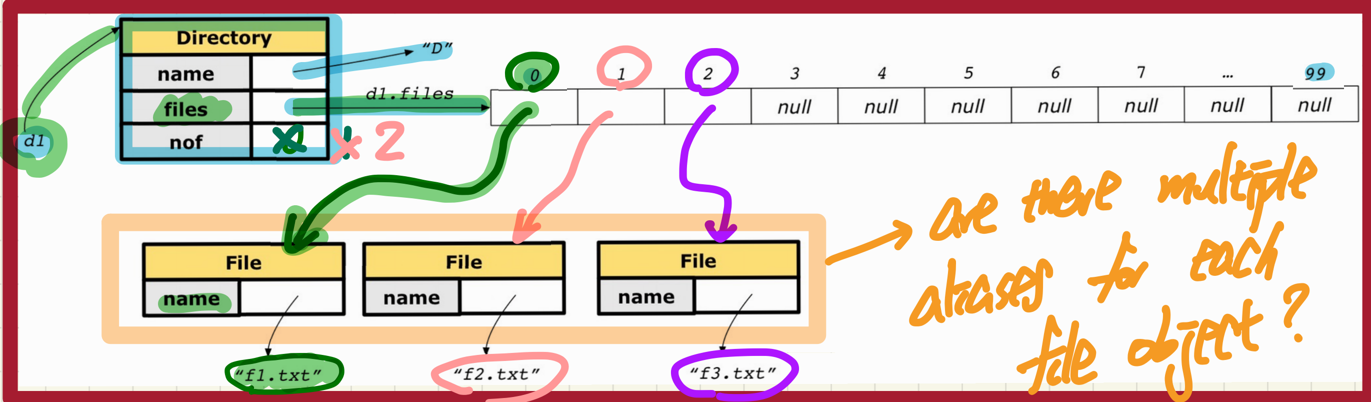
```
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(String name) {
    this.name = name;
    files = new File[100];
  }
  void addFile(String fileName) {
    files[nof] = new File(fileName);
    nof ++;
  }
}
```

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  }
```

So far, it's a composition.



are there multiple aliases for each file object?

No ⇒ no sharing of files.

Composition: Copy Constructor (Shallow Copy)

```java
@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")); }
```

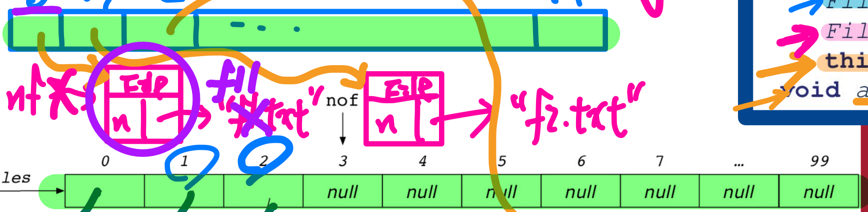
```java
class Directory {
  String name;
  File[] files;
  int nof; /* num of files */
  Directory(Directory other) {
    /* value copying for primitive type */
    nof = other.nof;
    /* address copying for reference type */
    name = other.name; files = other.files; }
```

Composition: Copy Constructor (Deep Copy)

```java
@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")); }
```

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

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

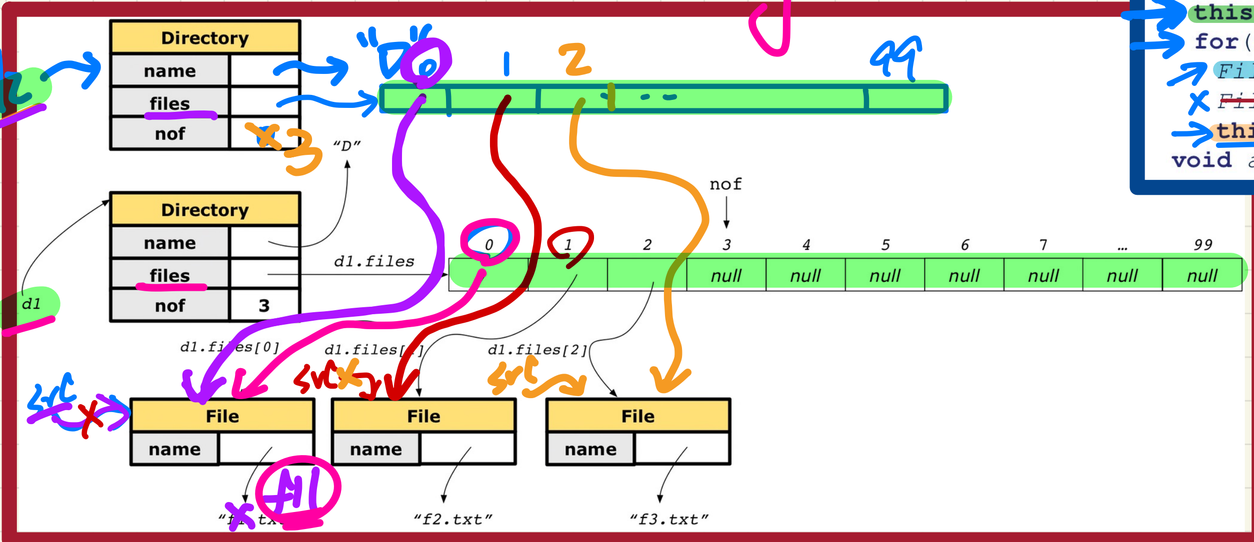


Exercise: Copy Constructor (Composition?)

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

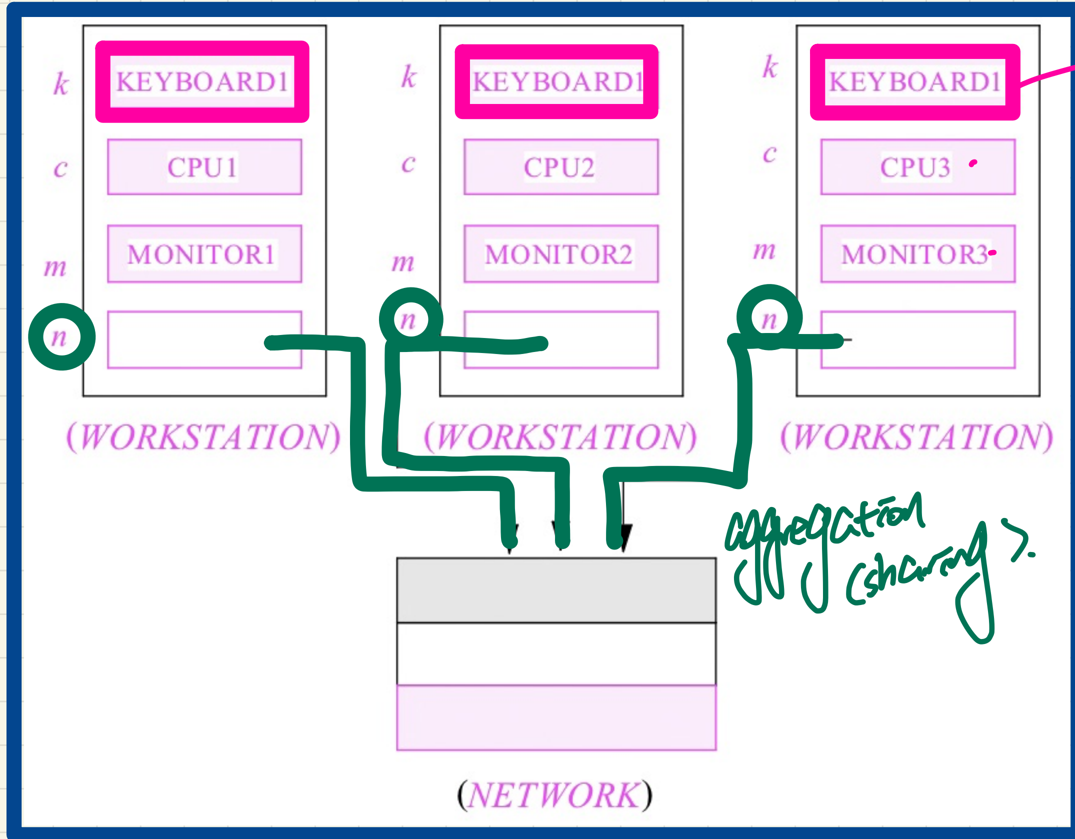


Lecture 4

Part G

Aggregation and Composition - Example and Exercise

Modelling: Aggregation vs. Composition

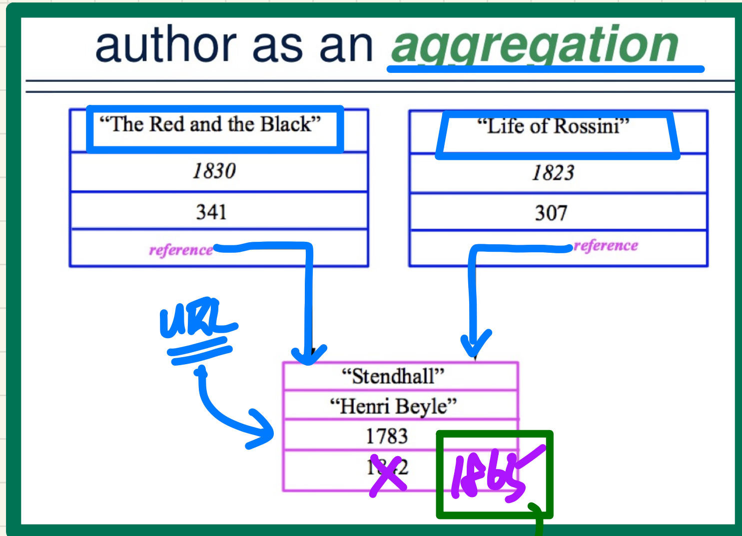


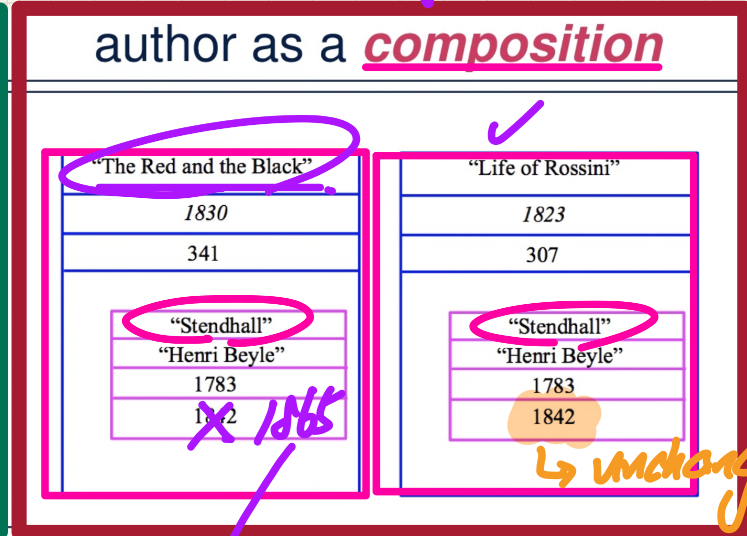
Composition

Exercise
↳ declare Java classes and attributes for implementing this design.

Implementation: Aggregation or Composition

What if the author field gets modified?





change only applied to its owning container.

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