RESCHEDULED:

FIRST GB MEETING & FALL INFO NIGHT!

LEARN ABOUT VOLUNTEERING ON THE SAMARITANS’ SUICIDE PREVENTION HOTLINE AND GETTING INVOLVED WITH THE CLUB!

THURSDAY, SEPT. 23RD
SALOMON 202, 7-8 PM

CHECK OUT OUR SOCIAL MEDIA FOR MORE INFO!

WEBSITE: SAMARITANSRI.ORG
FB: @STUDENTSFORSAMARITANS
IG: @STUDENTS.4.SAMARITANS
Hack@Brown applications are out now!
Visit tinyurl.com/habapply2022 to apply

Reach out to us at hello@hackatbrown.org if you have any questions!
Lecture 5

Working with Objects: Part 2
Outline

- Accessors and Mutators
- Association
  - “Many-to-One” Association
  - Component-Container Association
  - Two-way Association
Review: Variables

- Store information either as a value of a primitive or as a reference to an instance

```java
int favNumber = 9;
Dog django = new Dog();
<type> <name> = <value>;
```

Declaration: 
Initialization: 
Review: Instances as Parameters

- Methods can take in class instances as parameters
  
  ```java
  public void trimFur(Dog shaggyDog) {
  // code that trims the fur of shaggyDog
  }
  ```

- When calling the method above, every dog passed as an argument, e.g., `django`, will be thought of as `shaggyDog`, a synonym
Review: Variable Reassignment

● After giving a variable an initial value or reference, we can **reassign** it (make it store a different instance)

● When reassigning a variable, we do not declare its type again, Java remembers it from the first assignment

```java
Dog django = new Dog();
Dog scooby = new Dog();
django = scooby; // reassign django
```

● **django** now stores a different dog (another instance of Dog), specifically the one that was **scooby**. The initial dog stored by **django** is garbage collected
Review: Delegation Pattern

- Delegation allows us to separate different sets of functionalities and assign them to other classes.
- With delegation, we’ll use multiple classes to accomplish one task. A side effect of this is we need to set up relationships between classes for their instances to communicate.
- **Containment** is one of two key ways we establish these class relationships. We’ll learn the second one today. Stay tuned!
Review: Containment

- Often a class $A$ will need an instance of class $B$ as a *component*, so $A$ will create an instance of $B$ using the `new` keyword. We say $A$ contains an instance of class $B$
  - ex: `PetShop` creates a new `DogGroomer`
  - this is **not symmetrical**: $B$ can’t call methods on $A$!
- Ex: a `PetShop` can call methods of a contained `DogGroomer`, but the `DogGroomer` can’t call methods on the `PetShop`
- Containment is one of the ways we delegate responsibilities to other classes
Review: Local vs. Instance Variables (1/2)

- Local variables are declared inside a method and cannot be accessed from any other method.
- Once the method has finished executing, they are garbage collected.

```java
public class PetShop {
    // This is the constructor!
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.trimFur(django);
        django = new Dog();
        groomer.trimFur(django);
    }
}
```

Local Variables
Instance variables model properties or components that all instances of a class have

Instance variables are accessible from anywhere within the class – their scope is the entire class.

```java
public class PetShop {
    private DogGroomer groomer;
    public PetShop() {
        this.groomer = new DogGroomer();
        this.testGroomer();
    }
    // testGroomer elided
}
```
Review: Encapsulation

• In CS15, instance variables should be declared as `private`
• Why? **Encapsulation** for safety purposes
  o your properties are your private business
• If public, instance variables would be accessible from **anywhere**. There would be no way to restrict other classes from modifying them
• Private instance variables also allow for a chain of abstraction, so classes don’t need to worry about the inner workings of contained classes
• We’ll learn safe ways of allowing external classes to access instance variables
TopHat Question

Which of the following most accurately describes the containment relationships in this program?

A. App contains a Farm
B. App contains a House, a Pig, and multiple Cows
C. Farm contains a House, a Pig, and multiple Cows
D. A and C
E. A, B, and C

```java
public class App {
    public static void main(String[] args) {
        new Farm();
    }
}

public class Farm {
    private House farmHouse;
    private Pig wilbur;
    private Cow bessy;
    private Cow betty;
    public Farm() {
        this.farmHouse = new House();
        this.wilbur = new Pig();
        this.bessy = new Cow();
        this.betty = new Cow();
    }
}
```
TopHat Question

What visualization most accurately describes the containment relationships in this program?

Take a minute to sketch on your own, then we’ll show options on the next slide.

```java
public class App {
    public static void main(String[] args) {
        new Farm();
    }
}

public class Farm {
    private House farmHouse;
    private Pig wilbur;
    private Cow bessy;
    private Cow betty;

    public Farm() {
        this.farmHouse = new House();
        this.wilbur = new Pig();
        this.bessy = new Cow();
        this.betty = new Cow();
    }
}
```
TopHat Question

What visualization most accurately describes the containment relationships in the program?

A

B

C
Outline

- Accessors and Mutators
- Association
  - “Many-to-One” Association
  - Component-Container Association
  - Two-way Association
Accessors / Mutators

- A class may choose to allow other classes to have selective access to designated properties
  - e.g., Dog can allow DogGroomer to access its furlength property
- To do this, the class can make the value of an instance variable publicly available via an accessor method
- These accessor methods typically have the name convention get<Property> and have a non-void return type
- The return type specified and value returned must also match!
- Let’s see an example
Accessors / Mutators: Example

- Let’s make Dog’s furLength property publicly available

- getFurLength is an accessor method for furLength

- Can call getFurLength on an instance of Dog to return its current furLength value

- DogGroomer can now access this value. We will see why this is useful in a few slides

```java
public class Dog {
    private int furLength;
    public Dog() {
        this.furLength = 3;
    }

    public int getFurLength() {
        return this.furLength;
    }

    /* bark, eat, and wagtail elided */
}
```
Accessors / Mutators

• A class can give other classes even greater permission by allowing them to change the value of its instance variables
  ○ e.g., Dog can allow DogGroomer to change the value of its furLength property (i.e. instance variable)
• To do this, the class can define a mutator method which modifies the value of an instance variable
• These methods typically have the name convention set<Property> and have void return types
• They also take in a parameter that is used to modify the value of the instance variable
Accessors / Mutators: Example (1/6)

- Let’s define a mutator method, `setFurLength`, in `Dog` that sets `myFurLength` to the value passed in.
- `DogGroomer` can call `setFurLength` on an instance of `Dog` to change its `myFurLength` value.
- In fact, `DogGroomer` can use both `getFurLength` and `setFurLength` to modify `myFurLength` based on its previous value. Stay tuned for an example.

```java
public class Dog {
    private int myFurLength;

    public Dog() {
        this.myFurLength = 3;
    }

    public int getFurLength() {
        return this.myFurLength;
    }

    public void setFurLength(int furLength) {
        this.myFurLength = furLength;
    }

    /* bark, eat, and wagTail elided */
}
```
Accessors / Mutators: Example (2/6)

- Fill in `DogGroomer`'s `trimFur` method to modify the `furLength` of the `Dog` whose fur is being trimmed.

- When a `DogGroomer` trims the fur of a dog, it calls the `mutator` `setFurLength` on the `Dog` and passes in 1 as an argument. This will be the new value of `furLength`.

```java
public class DogGroomer {

    public DogGroomer() {
        // this is the constructor!
    }

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Accessors / Mutators: Example (3/6)

Check that `trimFur` works by printing out the `Dog's furLength` before and after we send it to the `groomer`.

```java
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        this.groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        System.out.println(django.getFurLength());
        this.groomer.trimFur(django);
        System.out.println(django.getFurLength());
    }
}
```

```java
public class DogGroomer {
    private DogGroomer() {
        // this is the constructor!
    }

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```

We use the accessor `getFurLength` to retrieve the value `django` stores in its `furLength` instance variable.
Accessors / Mutators: Example (4/6)

- What values print out to the console?

```java
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        this.groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        System.out.println(django.getFurLength());
        this.groomer.trimFur(django);
        System.out.println(django.getFurLength());
    }
}
```

```java
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```

- first, 3 is printed because that is the initial value we assigned to `furLength` in the `Dog` constructor (slide 13)
- next, 1 prints out because `groomer` just set `django`'s `furLength` to 1
Accessors / Mutators: Example (5/6)

- What if we don’t always want to trim a Dog’s fur to a value of 1?
- When we tell groomer to trimFur, let’s also tell groomer the length to trim the Dog’s fur

```java
public class PetShop {
    // Constructor elided
    public void testGroomer() {
        Dog django = new Dog();
        this.groomer.trimFur(django);
    }
}
```

```java
public class DogGroomer {
    /* Constructor and other code elided */
    public void trimFur(Dog shaggyDog, int furLength) {
        shaggyDog.setFurLength(furLength);
    }
}
```

- trimFur will take in a second parameter, and set Dog’s fur length to the passed-in value of furLength (note Dog doesn’t error check to make sure that furLength passed in is less than current value of furLength)
- Now pass in two arguments when calling trimFur so groomer knows how much furLength should be after trimming fur

The groomer will trim the fur to a furLength of 2!
Accessors / Mutators: Example (6/6)

- What if we wanted to make sure the value of `furLength` after trimming is always less than the value before?
- When we tell `groomer` the length to trim the `Dog`'s fur, let’s specify a length less than the current value of `furLength`

```java
public class PetShop {
    // Constructor elided
    public void testGroomer() {
        Dog django = new Dog();
        int newLen = this.groomer.getFurLength() - 2;
        this.groomer.trimFur(django, newLen);
    }
}

public class DogGroomer {
    /* Constructor and other code elided */
    public void trimFur(Dog shaggyDog, int furLength) {
        shaggyDog.setFurLength(furLength);
    }
}
```

- We could eliminate the local variable `newLen` by nesting a call to `getFurLength` as the second parameter:

```java
this.groomer.trimFur(django, this.groomer.getFurLength() - 2);
```
Summary of Accessors/Mutators

- Instance variables should always be declared `private` for safety reasons
  - but classes may want to offer useful functionality that allows access to selective properties (instance variables)
- If we made these instance variables `public`, any method could change them, i.e., with the caller in control of the inquiry or change – this is unsafe
- Instead the class can provide accessors/mutators (often in pairs, but not always) which give the class control over how the variable is queried or altered. For example, a mutator could do error-checking on the new value to make sure it is in range
- Also, an accessor needn’t be as simple as returning the value of a stored instance variable – it is just a method and can do arbitrary computation on one or more variables
TopHat Question

Which of the following signatures is correct for accessor methods?

A) `public void getFarmHouse() {
   return this.farmhouse;
}

B) `public House getFarmHouse() {
   return this.farmhouse;
}

C) `public House getFarmHouse(FarmHouse myFarmHouse) {
   this.farmhouse = myFarmhouse;
}

D) `public House getFarmHouse(FarmHouse myFarmHouse) {
   return this.myFarmHouse;
}

```java
public class Farm {
   private House farmhouse;

   // Farm constructor
   public Farm() {
      this.farmhouse = new House();
   }
}
```
Outline

- Accessors and Mutators
- Association
  - “Many-to-One” Association
  - Association with container
  - Two-way Association
Association

• We’ve seen how to use containment to call methods on instances of a class
  o PetShop can create a new instance of DogGroomer to call methods on it
  o we think of the DogGroomer instance as a component of the PetShop
  o similarly, an Engine instance is a component of a Car

• Sometimes a class A cannot create an instance of another class B but needs to call methods on it
  o for example, HTAs don’t create TAs but need to communicate with them
  o how can A and B communicate so that A can call methods on an instance of B?

• We can pass an instance of B into the constructor of class A as an argument so that A now “knows about” B. This is non-symmetric: B doesn’t automatically know about A. We say that A and B are associated with each other
  o we can make association symmetric by separately telling B to be associated with A
Outline

- Accessors and Mutators
- Association
  - "Many-to-One" Association
  - Component-Container Association
  - Two-way Association
“Many-to-One” Association

- Multiple classes, say A and B, may need to communicate with the same instance of another class, say C, to accomplish a task. Let’s revisit our PetShop example from last lecture.

- Our goal is to set up a system that allows PetShop employees, in this case DogGroomer, to log in hours worked and the Manager to approve worked hours and make necessary payment.

- Manager can keep track of the DogGroomer’s worked hours in its class, in addition to its other functionalities.

- Alternatively, the Manager can delegate these tasks to another class.
  - doesn’t need to know how employee’s working hours are tracked as long as they are tracked.

- DogGroomer and Manager would need to “know about” this class in order to send messages to its instance.

- We’re adding complexity to our design by adding another class, but making the Manager less complex – like many things in life, it is a tradeoff!
“Many-to-One” Association

- If we define a `TimeKeeper` class as this third class, both the `DogGroomer` and `Manager` need to be associated with the same instance of `TimeKeeper`.

  Log in Hours Worked | Get hours worked
  DogGroomer          | Manager

- What would happen if they didn’t?
Example: Motivation for Association (1/9)

- If `DogGroomer` and `Manager` were associated with different instances, our pipeline would fail!

- Still abstract? Let’s see how this looks like with code!
Example: Setting up Association  (2/9)

- Let’s create a simple TimeKeeper class and define some of its properties and capabilities

- setStartTime and setEndTime record the start and end times of a working period

- computeHoursWorked calculates amount of hours worked

```java
public class TimeKeeper {
    private Time start;
    private Time end;

    public TimeKeeper() {
        //initialize start and end to 0
    }

    public void setStartTime(Time time) {
        this.start = time;
    }

    public void setEndTime(Time time) {
        this.end = time;
    }

    public Time computeHoursWorked() {
        return this.end - this.start;
    }
}
```
Example: Setting up Association (3/9)

- **DogGroomer** needs to send messages to an instance of **TimeKeeper** in order to keep track of their worked hours.

- Thus, set up an **association** between **DogGroomer** and **TimeKeeper**.

- Modify **DogGroomer**’s constructor to take in a parameter of type **TimeKeeper**. The constructor will refer to it by the name **keeper**.

- **DogGroomer** now needs to track time spent trimming fur so call **TimeKeeper**’s **setStartTime** and **setEndTime** methods inside **trimFur**.

- Even though **DogGroomer** was passed an instance of **TimeKeeper** in its constructor, how can **DogGroomer**’s other methods access this instance?

```java
public class DogGroomer {
    public DogGroomer() {
        // code for constructor
    }
    public void trimFur(Dog shaggyDog) {
        // code to call setStartTime
        shaggyDog.setFurLength(1);
        // code to call setEndTime
    }
}
```
Example: Setting up Association (4/9)

- Modify `DogGroomer` to store its knowledge of `TimeKeeper`.

- Declare an instance variable `myKeeper` in `DogGroomer` and have constructor initialize it to the passed parameter.

- `myKeeper` now records the `keeper` instance passed to `DogGroomer`'s constructor, for use by its other methods.
  - A third use of instance variables, in addition to properties and components.

- Inside `trimFur`, can now tell `this.myKeeper` to record start and end time.
  - We use Java's built-in method `Instant.Now()`.

```java
public class DogGroomer {
    private TimeKeeper myKeeper;

    public DogGroomer(TimeKeeper keeper) {
        this.myKeeper = keeper;
    }

    public void trimFur(Dog shaggyDog) {
        this.myKeeper.setStartTime(Instant.Now());
        shaggyDog.setFurLength(1);
        this.myKeeper.setEndTime(Instant.Now());
    }
}
```
Example: Using the Association (5/9)

- Back in our PetShop class, we need to modify how we instantiate the DogGroomer
- What argument should we pass into the constructor of DogGroomer?
  - a new instance of TimeKeeper

```java
public class DogGroomer {
    private TimeKeeper myKeeper;

    public DogGroomer(TimeKeeper keeper) {
        this.myKeeper = keeper; // store the assoc.
    }
}

public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        this.groomer = new DogGroomer(new TimeKeeper());
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        this.groomer.trimFur(django);
    }
}
```
Example Cont.: Setting up Association (6/9)

- Remember that the Manager, who deals with payments, and the DogGroomer use the TimeKeeper as an intermediary.
- The Manager’s makePayment() needs to know the hours worked by the DogGroomer.
  - The TimeKeeper keeps track of such information with its properties (See slide 34).

```java
public class Manager {
    public Manager() {
        // this is the constructor!
    }

    public void makePayment() {
        // code elided!
    }
}
```
Example Cont.: Setting up Association (7/9)

- We can set up a second association so the Manager can retrieve information from the TimeKeeper as needed.

- Following the same pattern as with DogGroomer, modify the Manager’s constructor to take in an instance of the TimeKeeper class and record it in an instance variable.

```java
public class Manager {
    private TimeKeeper myKeeper;

    public Manager(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper;
    }

    public void makePayment() {
        // code elided!
    }
}
```
Example Cont.: Setting up Association (8/9)

- Call TimeKeeper’s computeHoursWorked method inside makePayment to compute the total number of hours worked by an employee and use that to calculate their total wages

```java
public class Manager {
    private TimeKeeper myKeeper;
    private int rate;

    public Manager(TimeKeeper keeper) {
        // initialize myKeeper and rate
    }

    public int makePayment() {
        int hrs = this.myKeeper.computeHoursWorked();
        int wages = hrs * this.rate;
        return wages;
    }
}
```
Example Cont.: Using the Association (9/9)

- Back in **PetShop** class, add a new instance of **Manager** and associate it with **TimeKeeper**
- **Manager** makes payment after **groomer** trims fur
- **Note:** **groomer** and **manager** refer to the same **TimeKeeper** instance

```java
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        TimeKeeper keeper = new TimeKeeper();
        this.groomer = new DogGroomer(keeper);
        Manager manager = new Manager(keeper);
        this.testGroomer();
        manager.makePayment();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        this.groomer.trimFur(django);
    }
}
```
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        TimeKeeper keeper = new TimeKeeper();
        Manager manager = new Manager(keeper);
        this.groomer = new DogGroomer(keeper);
        this.testGroomer();
        manager.makePayment();
    }

    // testGroomer elided

    PetShop's naming local variable keeper is completely arbitrary and independent of formal parameter names keeper in Manager and DogGroomer - pure coincidence!
}

private DogGroomer groomer;

public class Manager {
    private TimeKeeper myKeeper;

    public Manager(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper;
    }
}

public class DogGroomer {
    private TimeKeeper myKeeper;

    public DogGroomer(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper;
    }
}
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        TimeKeeper keeper = new TimeKeeper();
        Manager manager = new Manager(keeper);
        this.groomer = new DogGroomer(keeper);
        this.testGroomer();
        manager.makePayment();
    }
    // testGroomer elided
}

public class Manager {
    private TimeKeeper myKeeper;

    public Manager(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper;
    }
}

public class DogGroomer {
    private TimeKeeper myKeeper;

    public DogGroomer(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper
    }
}

Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop’s constructor initializes all its instance and local variables.
The PetShop instantiates a new TimeKeeper, Manager and DogGroomer, passing the same TimeKeeper instance in as an argument to the Manager’s and DogGroomer’s constructors.
Association: Under the Hood (4/5)

When the DogGroomer’s and Manager’s constructors are called, their parameter, keeper, points to the same TimeKeeper that was passed in as an argument by the caller, i.e., the PetShop.
public class PetShop {
    private DogGroomer groomer;

    public PetShop() {
        TimeKeeper keeper = new TimeKeeper();
        Manager manager = new Manager(keeper);
        this.groomer = new DogGroomer(keeper);
        this.testGroomer();
        manager.makePayment();
    }

    // methods elided
}

public class Manager {
    private TimeKeeper myKeeper;

    public Manager(TimeKeeper keeper) {
        // this is the constructor!
        this.myKeeper = keeper;
    }
}

public class DogGroomer {
    private TimeKeeper myKeeper;

    public DogGroomer(TimeKeeper keeper) {
        this.myKeeper = keeper;
    }

    // methods elided
}

Somewhere in memory...

DogGroomer and Manager set their myKeeper instance variable to point to the same TimeKeeper they received as an argument. Now they “know about” the same TimeKeeper and share the same properties.
Wrong Association

• If different instances of `TimeKeeper` are passed to the constructors of `Manager` and `DogGroomer`, the `DogGroomer` will still log their hours, but the `Manager` will not see any hours worked when `computeHoursWorked` is called.

• This is because `Manager` and `DogGroomer` would be sending messages to different `TimeKeepers`.

• And each of those `TimeKeepers` could have different hours.

• Let’s see what this looks like under the hood.
Wrong Association: Under the Hood

DogGroomer and Manager set their `myKeeper` instance variable to point to different instances of `TimeKeeper`. A change in one instance (e.g., when an instance variable changes) is not reflected in the other instance.
Visualizing Association

- The diagram above illustrates class relationships in our program. In CS15, we shall refer to this diagram as a **Containment/Association** diagram
Association as a Design Choice

- How we associate classes in our program is a design choice
  - if we had multiple employees in the PetShop, it would not make sense to pass the same instance of TimeKeeper to all employees. Why?
    - they would all modify the same start and end instance variables
    - the Manager would need to know which employee they are paying
  - in such a case, we may choose to associate the Manager with the employees (each employee instance would have its own start and end variables that they can modify)

- In later assignments, you will have to justify your design choices and how you decide to associate your classes, if at all, would be one of them
TopHat Question

Which of the following lines of code would NOT produce a compiler error, assuming it’s written in the App class?

A  Farmer farmer = new Farmer(this);

B  Farmer farmer = new Farmer();

C  Distributor dist = new Distributor(new Farmer());

D  Farmer farmer = new Farmer(new Distributor());
Outline

- Accessors and Mutators
- Association
  - “Many-to-One” Association
  - Component-Container Association
  - Two-way Association
Component-Container Association

- We’ve shown how association can enable multiple classes to communicate with the same instance of another class
  - DogGroomer and Manager know about the same instance of TimeKeeper
- Another common use of association is when a component needs to communicate with its container
- Let’s say a class A creates an instance of class B. A can send messages to B but not vice versa. What if we wanted B to send messages to A?
  - we have to associate B with A
- Let’s look at an example
Example: Setting up Association (1/4)

Let’s write a program that models an orchestra

- define an Orchestra class which can contain different instrumentalists and the conductor

The play method will be used to start and direct the musical performance

The Conductor has the capabilities to do this so an instance of Conductor is contained in Orchestra. We say Conductor is a component of Orchestra

The Orchestra can tell the Conductor to start performance because it created it as a component

This is another example of delegation: from the Orchestra to the Conductor

```java
public class Orchestra {

    private Conductor conductor;

    public Orchestra() {
        //this is the constructor
        this.conductor = new Conductor();
        this.play();
    }

    public void play() {
        this.conductor.startPerformance();
    }

}
```
Example: Motivation for Association (2/4)

- But what if the **Conductor** needs to call methods on the **Orchestra**?
  - the conductor probably needs to know several things about the orchestra. E.g., how many instrumentalists are there? Which ones are present?

- We can set up an **association** so the **Conductor** can communicate with the **Orchestra**

- As before, we have to modify the **Conductor**’s constructor to take an **Orchestra** parameter
  - and record it in an instance variable

```java
public class Conductor {
    private Orchestra myOrchestra;

    public Conductor(Orchestra orchestra) {
        this.myOrchestra = orchestra;
    }

    public void startPerformance() {
        // code elided
    }

    // other methods elided
}
```
Example: Using the Association (3/4)

- Back in the **Orchestra** class, what argument should **Conductor**’s constructor store?
  - the **Orchestra** instance that created the **Conductor**

- How?
  - by passing **this** as the argument
    - i.e., the **Orchestra** tells the **Conductor** about itself

```java
public class Orchestra {
    private Conductor conductor;

    public Orchestra() {
        //this is the constructor
        this.conductor = new Conductor(this);
    }

    public void play() {
        this.conductor.startPerformance();
    }

    // other methods elided
}
```
Example: Using the Association (4/4)

- The instance variable, `myOrchestra`, stores the instance of `Orchestra` that the `Conductor` belongs to

- `myOrchestra` points to same `Orchestra` instance passed to the `Conductor`'s constructor

- After constructor has been executed and can no longer reference `orchestra`, any `Conductor` method can still access same `Orchestra` instance by the name `myOrchestra`
  - thus can call `bow` on `myOrchestra` in `endPerformance`

```java
public class Conductor {
    private Orchestra myOrchestra;

    public Conductor(Orchestra orchestra) {
        this.myOrchestra = orchestra;
    }

    public void startPerformance() {
        // code elided
    }

    public void endPerformance() {
        this.myOrchestra.bow();
    }
}
```
Containment/Association Diagram

Orchestra

Conductor

“contains one instance of”
“contains more than one instance of”
“knows about”
TopHat Question

Which of the following statements is correct, given the code below that establishes an association from Teacher to School?

```java
public class School {
    private Teacher myTeacher;

    public School() {
        this.myTeacher = new Teacher(this);
    }
    //additional methods, some using //this.myTeacher
}
```

```java
public class Teacher {
    private School mySchool;

    public Teacher(School school) {
        this.mySchool = school;
    }
    //additional methods, some using //this.mySchool
}
```

A. School can send messages to Teacher, but Teacher cannot send messages to School
B. Teacher can send messages to School, but School cannot send messages to Teacher
C. School can send messages to Teacher, and Teacher can send messages to School
D. Neither School nor Teacher can send messages to each other
TopHat Question Review

- Does `School` contain `Teacher`?
  - yes! `School` instantiated `Teacher`, therefore `School` contains a `Teacher`. `Teacher` is a component of `School`.

- Can `School` send messages to `Teacher`?
  - yes! `School` can send messages to all its components that it created

- Does `Teacher` contain `School`?
  - no! `Teacher` knows about `School` that created it, but does not contain it
  - but can send messages to `School` because it "knows about" `School`

```java
public class School {
    private Teacher teacher;

    public School() {
        this.myTeacher = new Teacher(this);
    }
    //additional methods, some using //this.myTeacher
}

public class Teacher {
    private School mySchool;

    public Teacher(School school) {
        this.mySchool = school;
    }
    //additional methods, some using //this.mySchool
}
Outline

- Accessors and Mutators
- Association
  - Association with intermediary
  - Component-Container Association
  - Two-way Association
Two-way Association

● In the previous example, we showed how two classes can communicate with each other
  ○ class A contains an instance of class B, thus can send messages to it
  ○ class B knows about its container, class A, thus can send messages to it too
● Sometimes, we may want to model peer classes, say, A and B, where neither is a component of the other
● If we want these classes to communicate with each other (no intermediate class necessary), we can set up a two-way association where class A knows about B and vice versa
● Let’s see an example
Example: Motivation for Association (1/10)

- Here we have the class `CS15Professor`
- We want `CS15Professor` to know about his Head TAs — he didn’t create them or vice versa, they are peers (i.e., no containment)
- And we also want Head TAs to know about `CS15Professor`
- Let’s set up associations!

```java
public class CS15Professor {
    // declare instance variables here
    // and here...
    // and here...
    // and here!

    public CS15Professor(/* parameters */) {
        // initialize instance variables!
        // ...
        // ...
        // ...
    }

    /* additional methods elided */
}
```
Example: Motivation for Association (2/10)

- The **CS15Professor** needs to know about 5 Head TAs, all of whom will be instances of the class **HeadTA**

- Once he knows about them, he can call methods of the class **HeadTA** on them: **remindHeadTA**, **setUpLecture**, etc.

- Take a minute and try to fill in this class

```java
public class CS15Professor {
    // declare instance variables here
    // and here...
    // and here...
    // and here!

    public CS15Professor(/* parameters */) {
        // initialize instance variables!
        // ...
        // ...
        // ...
    }

    /* additional methods elided */
}
```
Example: Setting up Association (3/10)

- Our solution: we record passed-in HTAs
- Remember, you can choose your own names for the instance variables and parameters
- The **CS15Professor** can now send a message to one of his HTAs like this:

```java
class CS15Professor {
    private HeadTA hta1;
    private HeadTA hta2;
    private HeadTA hta3;
    private HeadTA hta4;
    private HeadTA hta5;

    public CS15Professor(HeadTA firstTA, HeadTA secondTA, HeadTA thirdTA, HeadTA fourthTA, HeadTA fifthTA) {
        this.hta1 = firstTA;
        this.hta2 = secondTA;
        this.hta3 = thirdTA;
        this.hta4 = fourthTA;
        this.hta5 = fifthTA;
    }

    this.hta2.setUpLecture();

    /* additional methods elided */
}
```
Example: Using the Association (4/10)

- We’ve got the **CS15Professor** class down
- Now let’s create a professor and head TAs from a class that contains all of them: **CS15App**
- Try and fill in this class!
  - you can assume that the **HTA** class takes no parameters in its constructor

```java
public class CS15App {
    // declare CS15Professor instance var
    // declare five HTA instance vars
    // ...
    // ...
    public CS15App() {
        // instantiate the professor!
        // ...
        // ...
        // instantiate the five HTAs
    }
}
```

- We've got the **CS15Professor** class down
- Now let's create a professor and head TAs from a class that contains all of them: **CS15App**
- Try and fill in this class!
  - you can assume that the **HTA** class takes no parameters in its constructor

```java
public class CS15App {
    // declare CS15Professor instance var
    // declare five HTA instance vars
    // ...
    // ...
    public CS15App() {
        // instantiate the professor!
        // ...
        // ...
        // instantiate the five HTAs
    }
}
```
Example: Using the Association (5/10)

- We declare andy, daniel, harriet, lila, uv, and will as instance variables - they are peers.
- In the constructor, we instantiate them.
- Since the constructor of CS15Professor takes in 5 HeadTAs, we pass in daniel, harriet, lila, uv, and will.

```java
public class CS15App {
    private CS15Professor andy;
    private HeadTA daniel;
    private HeadTA harriet;
    private HeadTA lila;
    private HeadTA uv;
    private HeadTA will;

    public CS15App() {
        this.daniel = new HeadTA();
        this.harriet = new HeadTA();
        this.lila = new HeadTA();
        this.uv = new HeadTA();
        this.will = new HeadTA();
        this.andy = new CS15Professor(this.daniel, this.harriet, this.lila, this.uv, this.will);
    }
}
```
Example: Using the Association (6/10)

```java
public class CS15App {
    private CS15Professor andy;
    private HeadTA daniel;
    private HeadTA harriet;
    private HeadTA lila;
    private HeadTA uv;
    private HeadTA will;

    public CS15App() {
        this.daniel = new HeadTA();
        this.harriet = new HeadTA();
        this.lila = new HeadTA();
        this.uv = new HeadTA();
        this.will = new HeadTA();
        this.andy = new CS15Professor(this.daniel,
                                      this.harriet, this.lila,
                                      this.uv, this.will);
    }
}

public class CS15Professor {
    private HeadTA hta1;
    private HeadTA hta2;
    private HeadTA hta3;
    private HeadTA hta4;
    private HeadTA hta5;

    public CS15Professor(HeadTA firstTA,
                         HeadTA secondTA, HeadTA thirdTA
                         HeadTA fourthTA, HeadTA fifthTA) {
        this.hta1 = firstTA;
        this.hta2 = secondTA;
        this.hta3 = thirdTA;
        this.hta4 = fourthTA;
        this.hta5 = fifthTA;
    }

    /* additional methods elided */
}
```
More Associations (7/10)

- Now the **CS15Professor** can call on the **HeadTAs** but can the **HeadTAs** call on the **CS15Professor** too?
- No! Need to set up another association
- Can we just do the same thing and pass `this.andy` as a parameter into each **HeadTAs** constructor?

```java
public class CS15App {
    private CS15Professor andy;
    private HeadTA daniel;
    private HeadTA harriet;
    private HeadTA lila;
    private HeadTA uv;
    private HeadTA will;

    public CS15App() {
        this.daniel = new HeadTA();
        this.harriet = new HeadTA();
        this.lila = new HeadTA();
        this.uv = new HeadTA();
        this.will = new HeadTA();
        this.andy = new CS15Professor(this.daniel,
                                      this.harriet, this.lila,
                                      this.uv, this.will);
    }
}
```
When we instantiate daniel, harriet, lila, uv, and will, we would like to use a modified HeadTA constructor that takes an argument, this.andy

But this.andy hasn’t been instantiated yet (will get a NullPointerException)! And we can’t initialize andy first because the HeadTAs haven’t been created yet...

How to break this deadlock?

```java
class CS15App {
    private CS15Professor andy;
    private HeadTA daniel;
    private HeadTA harriet;
    private HeadTA lila;
    private HeadTA uv;
    private HeadTA will;

    public CS15App() {
        this.daniel = new HeadTA();
        this.harriet = new HeadTA();
        this.lila = new HeadTA();
        this.uv = new HeadTA();
        this.will = new HeadTA();
        this.andy = new CS15Professor(this.daniel, this.harriet, this.lila, this.uv, this.will);
    }
}
```
More Associations (9/10)

- To break this deadlock, we need to have a new mutator
- First, instantiate `daniel`, `harriet`, `lila`, `uv`, and `will`, then instantiate `andy`
- Use a new mutator, `setProf`, and pass `andy` to each `HeadTA` to record the association

```java
public class CS15App {
    private CS15Professor andy;
    private HeadTA daniel;
    private HeadTA harriet;
    private HeadTA lila;
    private HeadTA uv;
    private HeadTA will;

    public CS15App() {
        this.daniel = new HeadTA();
        this.harriet = new HeadTA();
        this.lila = new HeadTA();
        this.uv = new HeadTA();
        this.will = new HeadTA();
        this.andy = new CS15Professor(this.daniel, this.harriet, this.lila, this.uv, this.will);
        daniel.setProf(this.andy);
        harriet.setProf(this.andy);
        lila.setProf(this.andy);
        uv.setProf(this.andy);
        will.setProf(this.andy);
    }
}
```
More Associations (10/10)

public class HeadTA {
    private CS15Professor professor;
    public HeadTA() {
        //other code elided
    }
    public void setProf(CS15Professor prof) {
        this.professor = prof;
    }
}

• Now each HeadTA will know about andy!
More Associations

- But what happens if `setProf` is never called?
- Will the Head TAs be able to call methods on the `CS15Professor`?
- No! We would get a `NullPointerException`!
Containment/Association Diagram

- CS15App
- CS15Professor
- HeadTA

- "contains one instance of"
- "contains more than one instance of"
- "knows about"
Summary

Important Concepts:

• In OOP, it’s necessary for classes to interact with each other to accomplish specific tasks
• Delegation allows us to have multiple classes and specify how their instances can relate with each other. We’ve seen two ways to establish these relationships:
  o containment, where one class creates an instance of another (its component) and can therefore send messages to it
  o association, where one class knows about an instance of another class (that is not its component) and call methods on it
• Delegation is the first design pattern we’ve learnt in CS15. Stay tuned for a second design pattern coming up in the next lecture and more discussions about design later in the course.
Announcements

• Pong comes out today!
  o Due Monday 9/27 at 11:59 PM EST
  o No early or late hand in!

• HTA Hours
  o Fridays 3:00 – 4:00 PM at Friedman 101

• Section Swaps
  o Deadline to make permanent swaps Friday 09/24

• Mentorship Form comes out today!
Topics in Socially Responsible Computing

Venture Capital, Startups, and Monopolies

(none of this constitutes investment advice)
Startups

• Many definitions!
• Ours: for profits and non-profits
  o small initial group of people (the “founders”)
  o working on a totally new idea or significant variation on old ideas
  o “unfair competitive advantage” / “unique value proposition” — secret sauce!
Startup funding models

• “Bootstrapping”
  o funded (at least initially) without formal investors
  o sometimes friends + family rounds, scrappy + extremely efficient w/ money

• Nonprofit startups!
  o small team but big impact—mission beyond profit!
  o examples: Ameelio, Upsolve, Signal (more later in course!)

Initially bootstrapped companies:

Nonprofit tech startups:

(from left to right, top to bottom): Squarespace, MailChimp, Atlassian, WayFair, Grammarly, GitHub, Patagonia

(from left to right, top to bottom): GoFundMe, Develop For Good, Wikipedia, Mozilla, Signal
What is venture capital?

• Funding model!

• VC firms/Funds: Sequoia, a16z, Kleiner Perkins, Bessemer, etc (Sand Hill Road, Palo Alto)

• They have investors themselves! (“limited partners”)
  o Pension funds
  o Wealthy investors
  o Companies
  o Universities (like Brown!)

• High risk, high (potential) payoff
Venture Capital (an oversimplification)

Investing in Publicly Traded Cos

$1  $1  $1  $2  $2  $2  $2  $2  $2

$2  $2  $2  $2  $2  $2  $2  $2  $2

Venture Capital

$1  $1  $1  $1  $1  $1  $1  $1  $1

$1  $1  $1  $1  $1  $1  $1  $1  $1

$20
Why take venture capital money?

• Scale first, capture the market, extract value later
• How important is time to market? Beat competitors to releasing a product (“prime mover advantage”)
• Looking for the version of your idea that could be huge!
• Raise money for capital-intensive projects that couldn’t get funded otherwise

Reid Hoffman’s “Blitzscaling,” Photo Credit: Amazon
Pros of Startups

• Grow really quickly, maximize impact
• Testing ground for ideas that may not be immediately profitable—innovate!
• Generally fewer people, easier to change company culture / influence company decisions
  o “on the ground floor”
  o still subject to market forces, decisions by investors, etc

Apple was successful at this point (pre-Mac products), but still small/a startup. The original Mac team. Photo Credit: Apple
Cons of Venture Capital/Startups (1/3)

• Extreme push for scale: limited set of problems one can solve with a business model that seeks rapid growth / bigness
  ◦ “When I was raising a round for a company I was working on, I was asked a common question, ‘What is the billion-dollar version of your idea?’” — Jamie Wang, “Value Beyond Instrumentalization”

• Scaling is hard work + strains workers

• “Minimum viable product” — what is the least we can build to validate that people want our idea?
Cons of Venture Capital/Startups (2/3)

• Sacrifices on accessibility, edge cases

• Founders’ ownership is diluted with each successive round of financing

• Ceding control: “VCs often get in too early, eat you alive, and exercise disproportionate control” — Andy (part of 20 startups!)

Mark Zuckerberg at the F8 conference in 2014. Photo Credit: Mike Deerkoski, Wikimedia
Cons of Venture Capital/Startups (3/3)

Venture capital pushes for (initially) unprofitable growth
- unfair competition
- often companies never become profitable!
More reading that may be of interest!

- Startup = Growth — Paul Graham
- “How Venture Capitalists are Deforming Capitalism” — Charles Duhigg, The New Yorker
- Voices of the Valley — Ben Tamoff & Moira Wegel, Logic x FSG
- “Beyond Instrumentalization” — Jamie Wang, Kernel Magazine
- Zero to One — Peter Thiel, Crown Business
- Abolish Silicon Valley — Wendy Liu, Repeater
- Blitzscaling — Reid Hoffman, Crown Business
- What Tech Calls Thinking — Adrian Daub, Logic x FSG
- Secrets of Sand Hill Road — Scott Kupor, Penguin
- venturecommune.substack.com — Jason Prado
- The Pmarca guide to startups — Marc Andreessen
- Platform Capitalism — Nick Srnicek, Wiley
- The Memos — Bessemer Venture Partners
- Tech & VC: The Foundation — Paige Finn Doherty