Lecture 5

Working with Objects: Part 2

Review Topics at the end of the deck

Please make sure you understand what we have covered so far

- Variables
- Local vs. Instance Variables
- Variable Reassignment
- Instances as Parameters
- Delegation Pattern
- NullPointer Exceptions
- Encapsulation
- Containment

TopHat Question

Join Code: 504547

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 `Cow`s
C. `Farm` contains a `House`, a `Pig`, and multiple `Cow`s
D. A and C
E. A, B, and C

```
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

Join Code: 504547

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

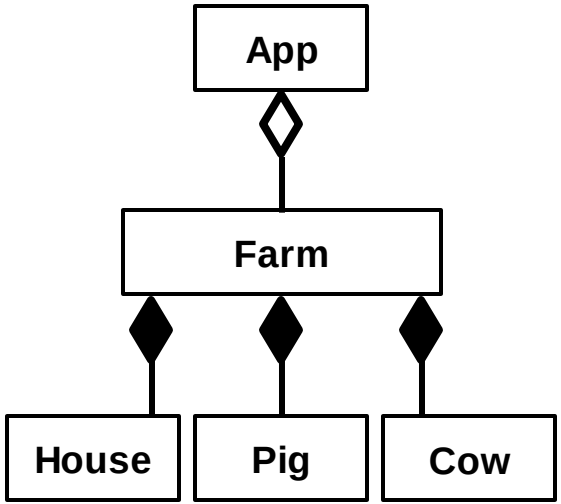
Take a minute to sketch on your own, then we'll show options on the next slide.

```
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();
    }
}
```

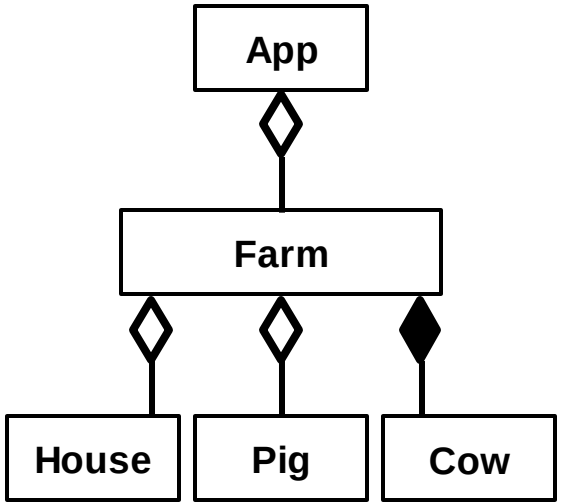
TopHat Question

Join Code: 504547

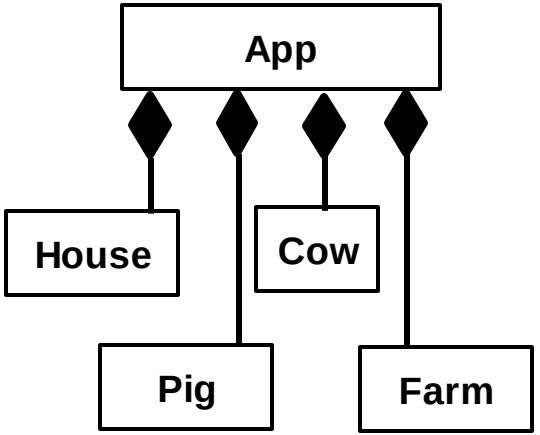
What visualization most accurately describes the containment relationships in the program?



A



B



C

Outline

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

Accessors / Mutators

- All instances of a class have the same instance variables (properties) but their own values
- Instance variables hold the instance's `private` properties: encapsulation
- But 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

```
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 properties/instance variables
  - e.g., `Dog` can allow `DogGroomer` to change the value of its `furLength` property
- 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 `furLength` to the value passed in
- `DogGroomer` can call `setFurLength` on an instance of `Dog` to change its `furLength` value
- In fact, `DogGroomer` can use both `getFurLength` and `setFurLength` to modify `furLength` based on its previous value. Stay tuned for an example

```
public class Dog {

   private int furLength;

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

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

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

```
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`

```
public class PetShop {
   private DogGroomer groomer;

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

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

```
public class DogGroomer {

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

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

We use the **accessor** `getFurLength` to retrieve the value `effie` stores in its `furLength` instance variable

Accessors / Mutators: Example (4/6)

- What values print out to the console?

Code from previous slide!

```
public class PetShop {
   private DogGroomer groomer;

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

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

```
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 10)
- next, 1 prints out because `groomer` just set `effie`'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

```
public class PetShop {
   // Constructor elided
   public void testGroomer() {
      Dog effie = new Dog();
      this.groomer.trimFur(effie, 2);
   }
}
```

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

The groomer will trim the fur to a furLength of 2!

- `trimFur` will take in a second parameter, and set `Dog`'s fur length to the passed-in value of `furLength` (note for simplicity `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

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`

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

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

decrease furLength by 2

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

```
this.groomer.trimFur(effie, effie.getFurLength() - 2);
```

Summary of Accessors/Mutators

- Instance variables should always be declared `private` for safety reasons
- 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
- Use them sparingly – only when other classes need them

TopHat Question

Join Code: 504547

Which of the following signatures is correct for an accessor method in Farm?

```
public class Farm {
    private House farmHouse;

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

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;
}
```

Outline

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

Association

- We've seen how a container instance can call methods on any contained instances it "**new**ed", but this relationship is not symmetric: the contained instance cannot communicate with its container!
  - `Orchestra` creates a **new** instance of a `Conductor`
  - The `Conductor` instance is a *component* of the `Orchestra`
  - The `Orchestra` can now call methods on the `Conductor`
  - But what if the `Conductor` needs to communicate with the `Orchestra`?
  - We need additional code to allow this symmetry
- We will tell the Conductor about the instance that created it, in this case, an Orchestra instance. We want to **associate** the Conductor with the Orchestra
  - The easiest way is to pass the `Orchestra` instance as a parameter to the `Conductor`'s constructor
  - How?!?

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`

```
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? When is the next rehearsal?...
- We can set up an association so the `Conductor` can communicate with the `Orchestra`
- We modify the `Conductor`'s constructor to take an `Orchestra` parameter
  - and record it in an instance variable
  - but where do we get this Orchestra?

```
public class Conductor {

   private Orchestra orchestra;
   // other instance variables elided

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

   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 be passed?
  - the `Orchestra` instance that created the `Conductor`
- How?
  - by passing `this` as the argument
    - i.e., the `Orchestra` tells the `Conductor` about itself

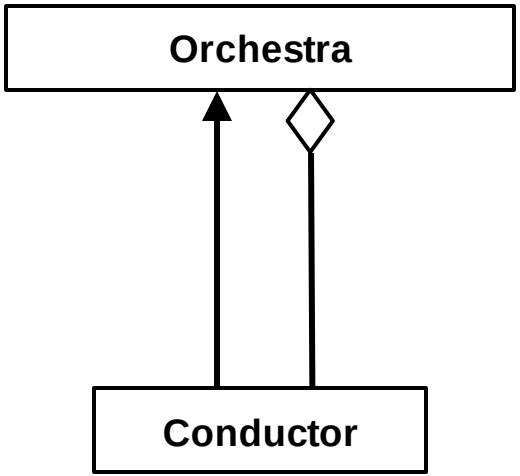
```
public class Orchestra {
   private Conductor conductor;
   // other instance variables elided

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

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

   // other methods elided
}
```

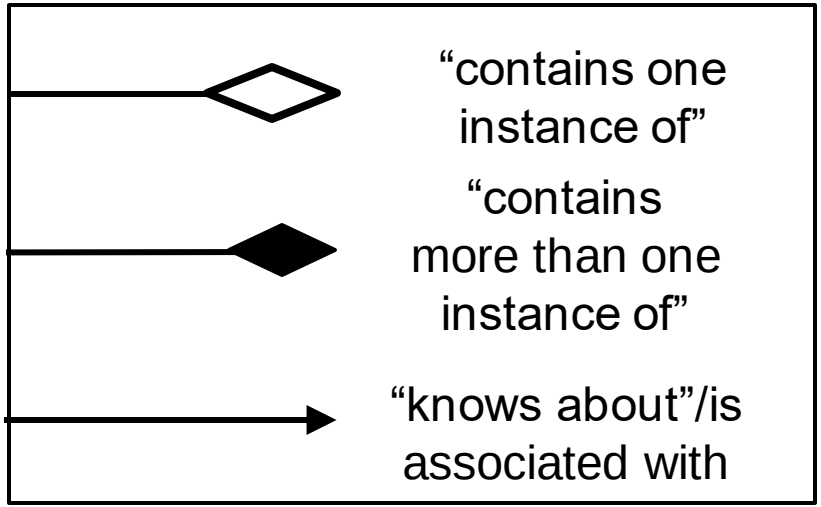
Example: Using the Association (4/4)

- The instance variable, `orchestra`, stores the instance of `Orchestra` of which the `Conductor` is a component
- `orchestra` points to same `Orchestra` instance passed to the `Conductor`'s constructor
- After constructor has been executed and can no longer reference **parameter** `myOrchestra`, any `Conductor` method can still access same `Orchestra` instance by the name `orchestra`
  - thus can call `bow` on `orchestra` in `endPerformance`

```
public class Conductor {
   private Orchestra orchestra;

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

   public void startPerformace() {
       // code elided
   }

   public void endPerformance() {
        this.orchestra.bow();
   }

}
```

Containment/Association Diagram





TopHat Question

Join Code: 504547

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

```
public class School {
  private Teacher teacher;

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

```
public class Teacher {
  private School school;

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

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

```
public class School{
  private Teacher teacher;

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

```
public class Teacher{
  private School school;

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

- 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`

Outline

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

“Many-to-One” Association

- Multiple classes, say `A` and `B`, may need to communicate with the same instance of another (peer) class, say `C`, to accomplish a task. Let’s consider our `PetShop` example
- 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, peer 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 weren’t associated?

Example: Motivation for Association (1/9)

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

`DogGroomer`

Log in Hours Worked

Get hours worked

`Manager`

- Still abstract? Let's see how this looks like with code!

Example: Motivation for 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

```
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: Motivation for 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 myKeeper
- DogGroomer now needs to track time spent trimming fur so call TimeKeeper's setStartTime and setEndTime methods inside the simple trimFur, the one that takes in just a Dog
- Even though DogGroomer was passed an instance of TimeKeeper in its constructor, how can DogGroomer's other methods access this instance?

```
public class DogGroomer {

   public DogGroomer(TimeKeeper myKeeper)
{     // code for constructor
   }  // code for modified constructor
   }
}
   public void trimFur(Dog shaggyDog) {
      // code to call setStartTime
      shaggyDog.setFurLength(1);
      // code to call setEndTime
   }
}
```

Example: Motivation for Association (4/9)

- Modify `DogGroomer` to store its knowledge of `TimeKeeper` in an instance variable
- Declare an instance variable `keeper` in `DogGroomer` and have constructor initialize it to the passed parameter
- `keeper` now records the `myKeeper` instance passed to `DogGroomer`'s constructor, for use by its other methods
- Inside `trimFur`, can now tell `this.keeper` to record start and end time
  - we use Java's built-in method `Instant.Now();`

```
public class DogGroomer {
   private TimeKeeper keeper;

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

   public void trimFur(Dog shaggyDog) {
      this.keeper.setStartTime(Instant.Now());
       shaggyDog.setFurLength(1);
       this.keeper.setEndTime(Instant.Now());
   }
}
```

Example: Motivation for Association (5/9)

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

```
public class DogGroomer {
   private TimeKeeper keeper;

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

public class PetShop {
   private DogGroomer groomer;

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

   public void testGroomer() {
      Dog effie = new Dog(); // local var
      this.groomer.trimFur(effie);
   }
}
```

Example Cont.: Setting up Association (6/9)

DogGroomer

Log in Hours Worked

Get hours worked

Manager

- 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 31)

```
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

```
public class Manager {
  private TimeKeeper keeper;

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

```
public class Manager {

  private TimeKeeper keeper;
  private int rate;

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

   public int makePayment() {
      int hrs = this.keeper.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

```
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 effie = new Dog();//local var
      this.groomer.trimFur(effie);
   }
}
```

Association: Under the Hood (1/5)

```
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 myKeeper in Manager and DogGroomer - pure coincidence!

```
public class Manager {
  private TimeKeeper keeper;

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

```
public class DogGroomer {
  private TimeKeeper keeper;

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

Somewhere in memory...

Association: Under the Hood (2/5)

```
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 keeper;

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

```
public class DogGroomer {
  private TimeKeeper keeper;

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

Somewhere in memory...

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

Association: Under the Hood (3/5)

```
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 keeper;

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

```
public class DogGroomer {
  private TimeKeeper keeper;

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

Somewhere in memory...

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)

```
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 keeper;

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

```
public class DogGroomer {
   private TimeKeeper keeper;

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

Somewhere in memory...

When the DogGroomer's and Manager's constructors are called, their parameter, myKeeper, points to the same TimeKeeper that was passed in as an argument by the caller, i.e., the PetShop

Association: Under the Hood (5/5)

```
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 keeper;

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

```
public class DogGroomer {
   private TimeKeeper keeper;

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

Somewhere in memory...

`DogGroomer` and Manager set their `keeper` 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 `TimeKeeper`s
- And each of those `TimeKeeper`s could have different hours
- Let's see what this looks like under the hood

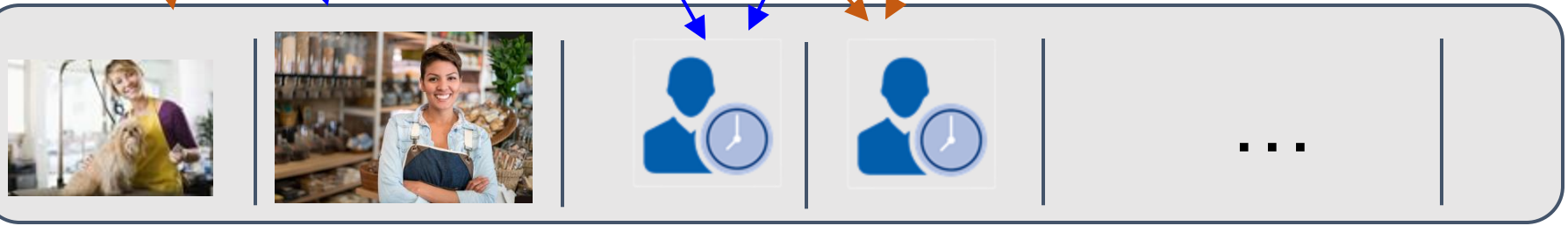
Wrong Association: Under the Hood

```
public class PetShop {
   private DogGroomer _groomer;

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

}
```

```
public class Manager {
  private TimeKeper keeper;

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

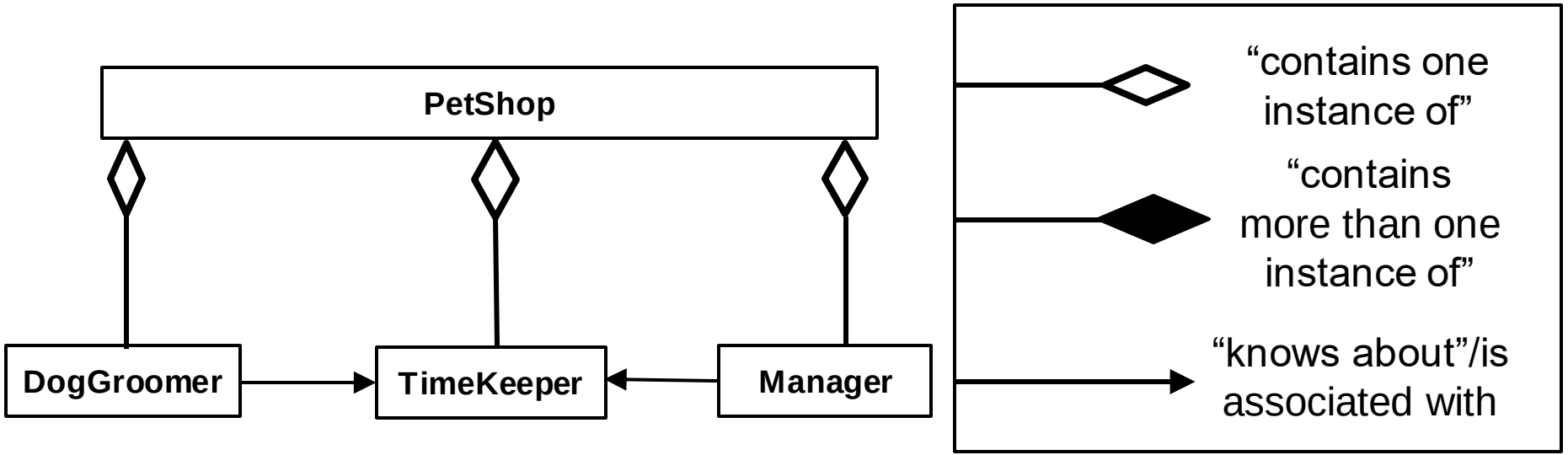
```
public class DogGroomer {
   private TimeKeeper keeper;

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

    }
}
```

Somewhere in memory...



DogGroomer and Manager set their keeper 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 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());
```

```
public class Farmer {
    private Distributor dist;

    public Farmer(Distributor myDist) {
      this.dist = myDist;
    }

}

public class Distributor {

    public Distributor() {

    }
 }
```

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 and we want the communication to be bidirectional

- 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!

```
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

```
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 created by whatever object creates CS15Professor and HTAs, e.g., CS15App
- Remember, you can choose your own names for the instance variables and parameters
- The `CS15Professor` can now send a message to one of his `HTA`s like this:

```
this.hta2.setUpLecture();
```

```
public class CS15Professor {

   private HTA hta1;
   private HTA hta2;
   private HTA hta3;
   private HTA hta4;
   private HTA hta5;

   public CS15Professor(HTA firstTA,
        HTA secondTA, HTA thirdTA,
        HTA fourthTA, HTA fifthTA) {

        this.hta1 = firstTA;
        this.hta2 = secondTA;
        this.hta3 = thirdTA;
        this.hta4 = fourthTA;
        this.hta5 = fifthTA;
   }

   /* 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

```
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`, `allie`, `anastasio`, `cannon`, `lexi`, and `sarah` as instance variables - they are peers
- In the constructor, we instantiate them
- Since the constructor of `CS15Professor` takes in 5 `HTA`s, we pass in `allie`, `anastasio`, `cannon`, `lexi`, and `sarah`

```
public class CS15App {
   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
       this.allie = new HTA();
       this.anastasio = new HTA();
       this.cannon = new HTA();
       this.lexi = new HTA();
       this.sarah = new HTA();
       this.andy = new
          CS15Professor(this.allie,
          this.anastasio, this.cannon,
                    this.lexi, this.sarah);
   }
}
```

Example: Using the Association (6/10)

```
public class CS15Professor {

   private HTA hta1;
   private HTA hta2;
   private HTA hta3;
   private HTA hta4;
   private HTA hta5;

   public CS15Professor(HTA firstTA,
          HTA secondTA, HTA thirdTA
          HTA fourthTA, HTA fifthTA) {

        this.hta1 = firstTA;
        this.hta2 = secondTA;
        this.hta3 = thirdTA;
        this.hta4 = fourthTA;
        this.hta5 = fifthTA;
   }

   /* additional methods elided */
}
```

```
public class CS15App {

   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
       this.allie = new HTA();
       this.anastasio = new HTA();
       this.cannon = new HTA();
       this.lexi = new HTA();
       this.sarah = new HTA();
       this.andy = new
          CS15Professor(this.allie,
          this.anastasio, this.cannon,
                   this.lexi, this.sarah);
   }
}
```

More Associations (7/10)

- Now the `CS15Professor` can call on the `HTA`s but can the `HTA`s 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 `HTA`s constructor?

```
public class CS15App {

   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
       this.allie = new HTA();
       this.anastasio = new HTA();
       this.cannon = new HTA();
       this.lexi = new HTA();
       this.sarah = new HTA();
       this.andy = new
           CS15Professor(this.allie,
           this.anastasio, this.cannon,
           this.lexi, this.sarah);
   }
}
```

Code from previous slide

More Associations (8/10)

- When we instantiate `allie`, `anastasio`, `cannon`, `lexi`, and `sarah`, we would like to use a modified `HTA` 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 `HTA`s haven't been created yet…
- How to break this deadlock?

```
public class CS15App {

   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
       this.allie = new HTA();
       this.anastasio = new HTA();
       this.cannon = new HTA();
       this.lexi = new HTA();
       this.sarah = new HTA();
       this.andy = new
           CS15Professor(this.allie,
           this.anastasio, this.cannon,
           this.lexi, this.sarah);
   }
}
```

Code from previous slide

More Associations (9/10)

- To break this deadlock, we need to have a new mutator
- First, instantiate `allie`, `anastasio`, `cannon`, `lexi`, and `sarah`, then instantiate `andy`
- Use a new mutator, `setProf`, and pass `andy` to each `HeadTA` to record the association

```
public class CS15App {

   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
      this.allie = new HTA();
      this.anastasio = new HTA();
      this.cannon = new HTA();
      this.lexi = new HTA();
      this.sarah = new HTA();
      this.andy = new CS15Professor(this.allie,
           this.anastasio, this.cannon, this.lexi,
           this.sarah);

      this.allie.setProf(this.andy);
      this.anastasio.setProf(this.andy);
      this.cannon.setProf(this.andy);
      this.lexi.setProf(this.andy);
      this.sarah.setProf(this.andy);        }
}
```

More Associations (10/10)

```
public class HTA {

   private CS15Professor professor;

   public HTA() {

      //other code elided

   }

   public void setProf(CS15Professor myProf)
   {
      this.professor = myProf;
   }
}
```

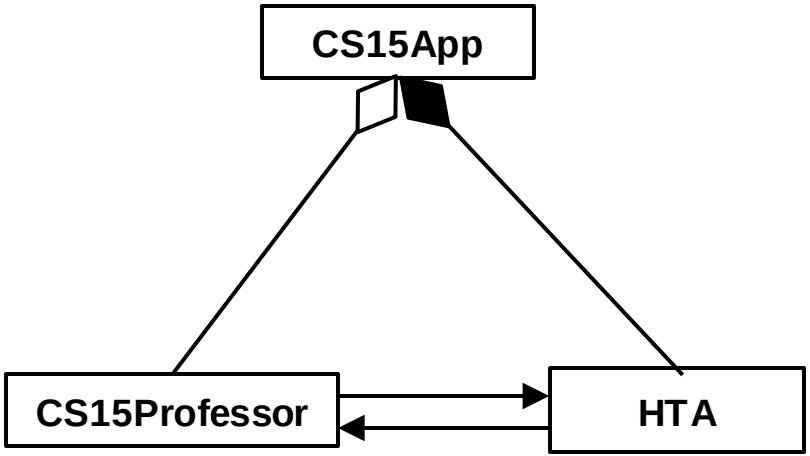
- Now each HTA will know about andy!

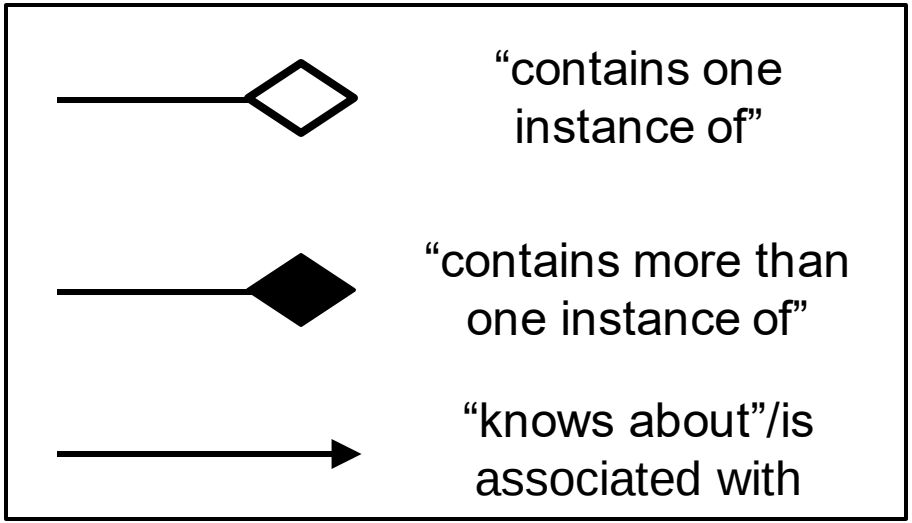
```
public class CS15App {

   private CS15Professor andy;
   private HTA allie;
   private HTA anastasio;
   private HTA cannon;
   private HTA lexi;
   private HTA sarah;

   public CS15App() {
      this.allie = new HTA();
      this.anastasio = new HTA();
      this.cannon = new HTA();
      this.lexi = new HTA();
      this.sarah = new HTA();
      this.andy = new CS15Professor(this.allie,
           this.anastasio, this.cannon, this.lexi,
           this.sarah);

      this.allie.setProf(this.andy);
      this.anastasio.setProf(this.andy);
      this.cannon.setProf(this.andy);
      this.lexi.setProf(this.andy);
      this.sarah.setProf(this.andy);        }
}
```

More Associations

- But what happens if `setProf` is never called?
- Will the HTAs be able to call methods on the `CS15Professor`?
- No! We would get a `NullPointerException`!
  - remember: `NullPointerExceptions` occur at **runtime** when a variable's value is null, and you try to give it a command

Containment/Association Diagram





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:
    - **containment**, where one class creates an instance of another (its component) and can therefore send messages to it
    - **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 learned 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!
  - Due Monday 9/25 at 11:59 PM EST
  - No early or late hand in!
- HTA Hours
  - Fridays 3:30 – 4:30 PM at CIT 210
- Section Swaps
  - Deadline to make permanent swaps Friday 09/22
- CS15 Mentorship!
  - Freshmen: It is mandatory for you to meet with your mentors. Please respond to their emails and be flexible.
  - If you have not gotten an assignment email the HTAs

Review: Variables

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

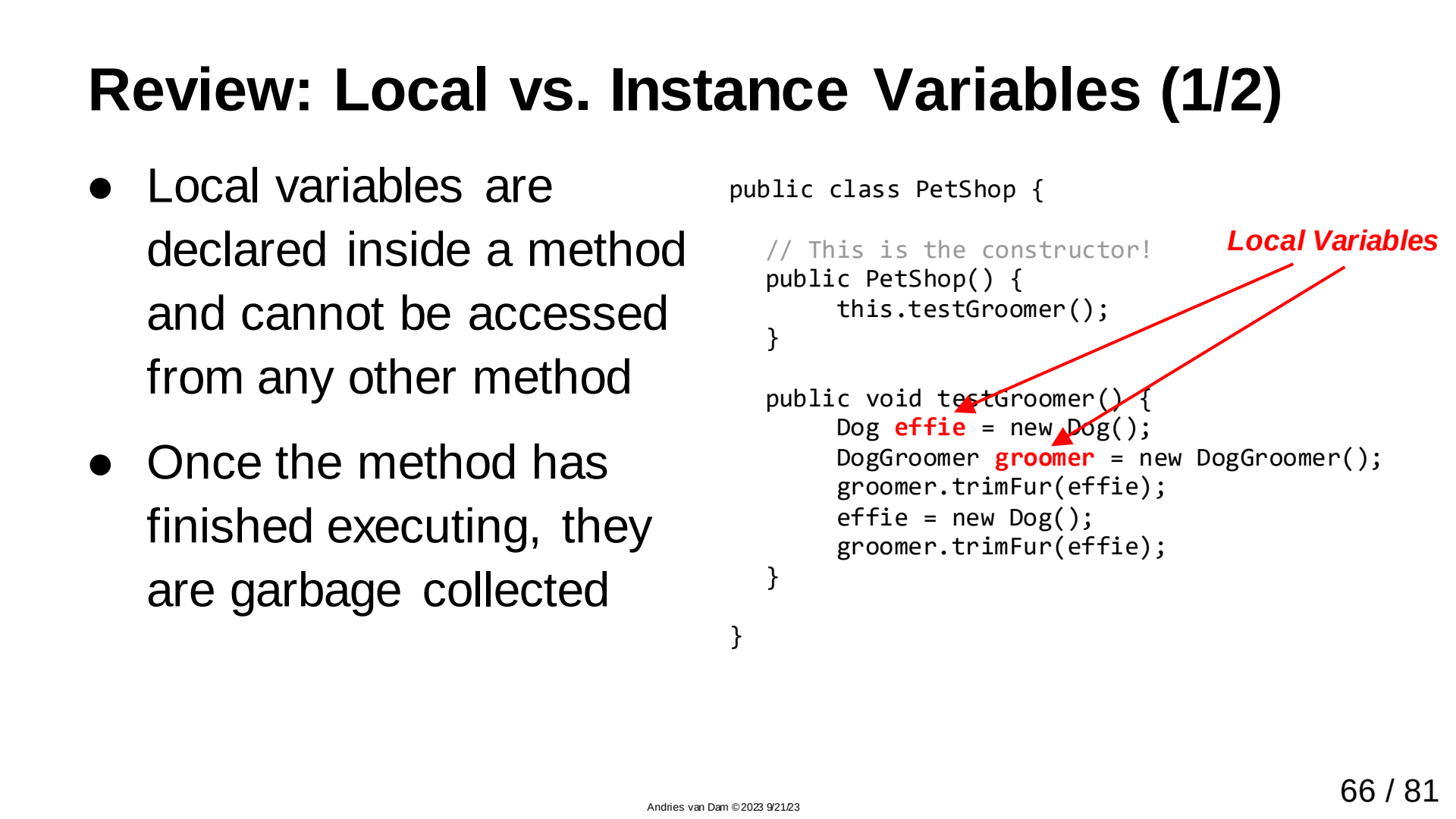
```
int favNumber = 9;

Dog effie = new Dog();

<type> <name> = <value>;
```

declaration initialization

Review: Local vs. Instance Variables (1/2)

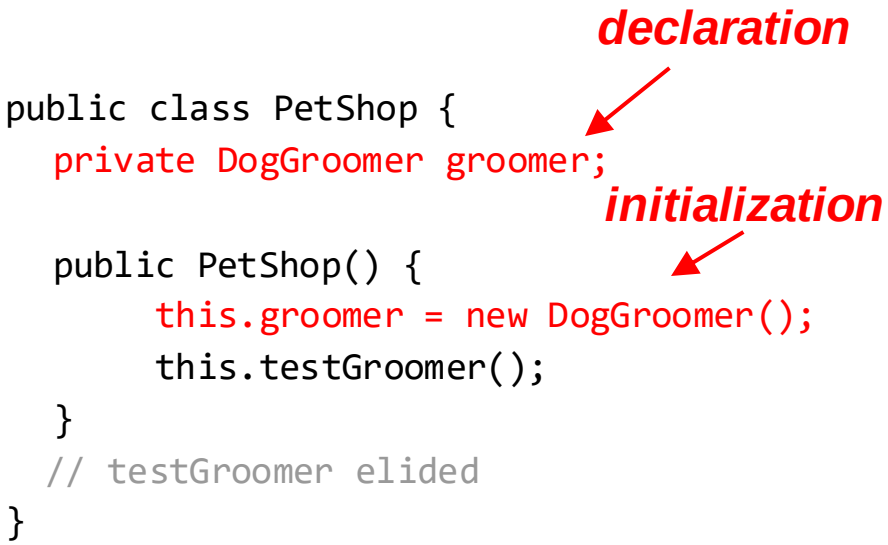
- Local variables are declared inside a method and cannot be accessed from any other method
- Once the method has finished executing, they are garbage collected

```
public class PetShop {

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

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

}
```



Review: Local vs. Instance Variables (2/2)

- 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
- The purpose of a constructor is to initialize all instance variables



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

```
Dog effie = new Dog();
Dog katniss = new Dog();

effie = katniss; // reassign effie
```

- `effie` now stores a different dog (another instance of Dog), specifically the one that was `katniss`. The initial dog stored by `effie` is garbage collected

Review: Instances as Parameters

- Methods can take in class instances as parameters

```
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., `effie`, will be thought of as `shaggyDog`, a synonym, in the method

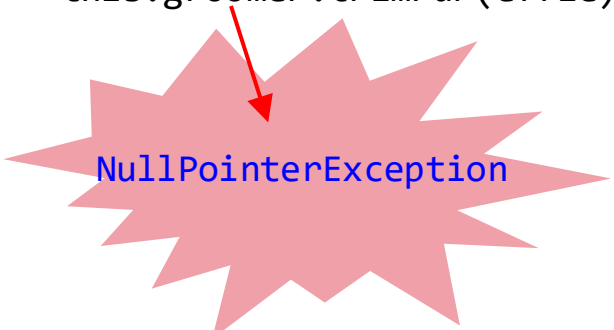
Review: Delegation Pattern

- 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: NullPointer Exceptions

- What happens if you fail to initialize an instance variable in the constructor?
  - instance variable `groomer` never initialized so default value is `null`
  - when a method is called on `groomer` we get a `NullPointerException`

```
public class PetShop {

   private DogGroomer groomer;

   public PetShop() {
       //oops! Forgot to initialize groomer
       this.testGrooming();
   }
   public void testGrooming() {
       Dog effie = new Dog(); //local var
       this.groomer.trimFur(effie);
   }
}
```



Review: Encapsulation

- In CS15, instance variables should be declared as `private`
- Why? **Encapsulation** for safety purposes
  - your properties are your private business
- If public, instance variables would be accessible from **any class**. 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

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`
  - ex: `Car` creates a `new Engine`
  - ex: `Body` creates a `new Head`
- 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`
    - a workaround uses assocation pattern
- Containment is one of the ways we delegate responsibilities to other classes

Topics in Socially Responsible Computing

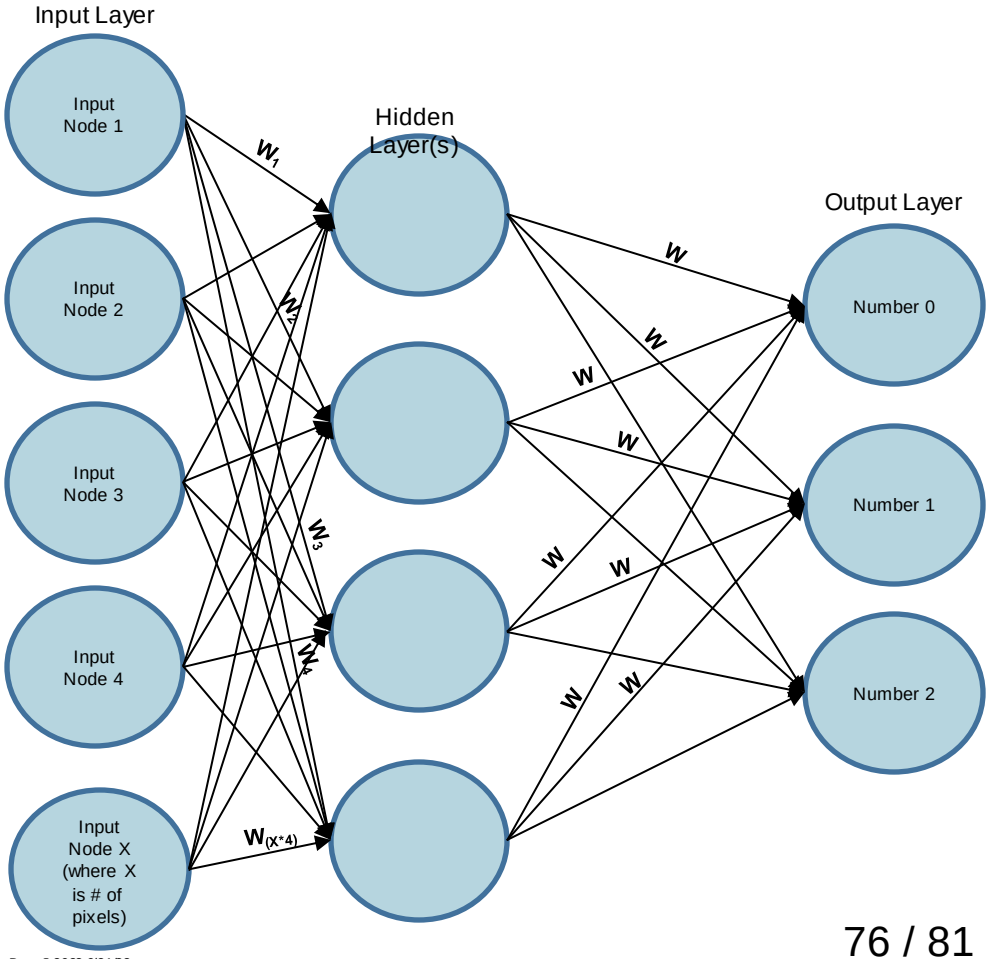
CS15 Fall 2023

Task: Image Recognition with Neural Network

- Neural networks are frequently used for image recognition
- ***Example: Given the following images, can our neural network identify which number is represented***

Neural Network Terminology

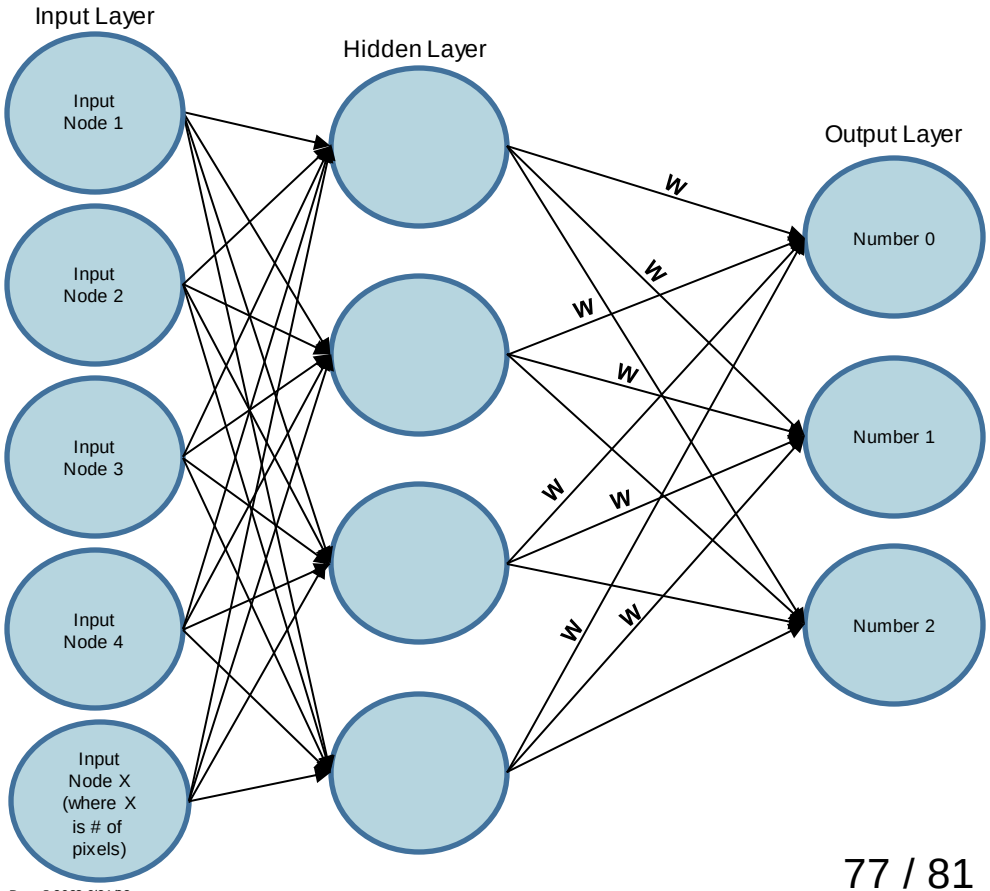
- **Nodes (also known as neuron or perceptron)**
  - A node is a highly simplified neuron
  - A node contains a value and stores data used for later calculations.
  - There should be an input node for each feature (in the case of image recognition, one input for each pixel)
- **Hidden Layer(s)**
  - Intermediate layer of nodes between the input and output
  - Transforms the input features so that they can be correctly classified in the output layer
- **Weights (also known as parameters)**



How Does a Feed Forward Neural Network Work? (1/2)

Step 1: Forward Pass

- **Receive inputs**
  - Initial data is passed in through the input layer
  - Ex. Each pixel from an image is an input for image recognition tasks.
- **Perform Computations**
  - For first pass weights are randomly initialized
  - At simplest level, each hidden layer takes a weighted sum of each input and their weights leading directly to it



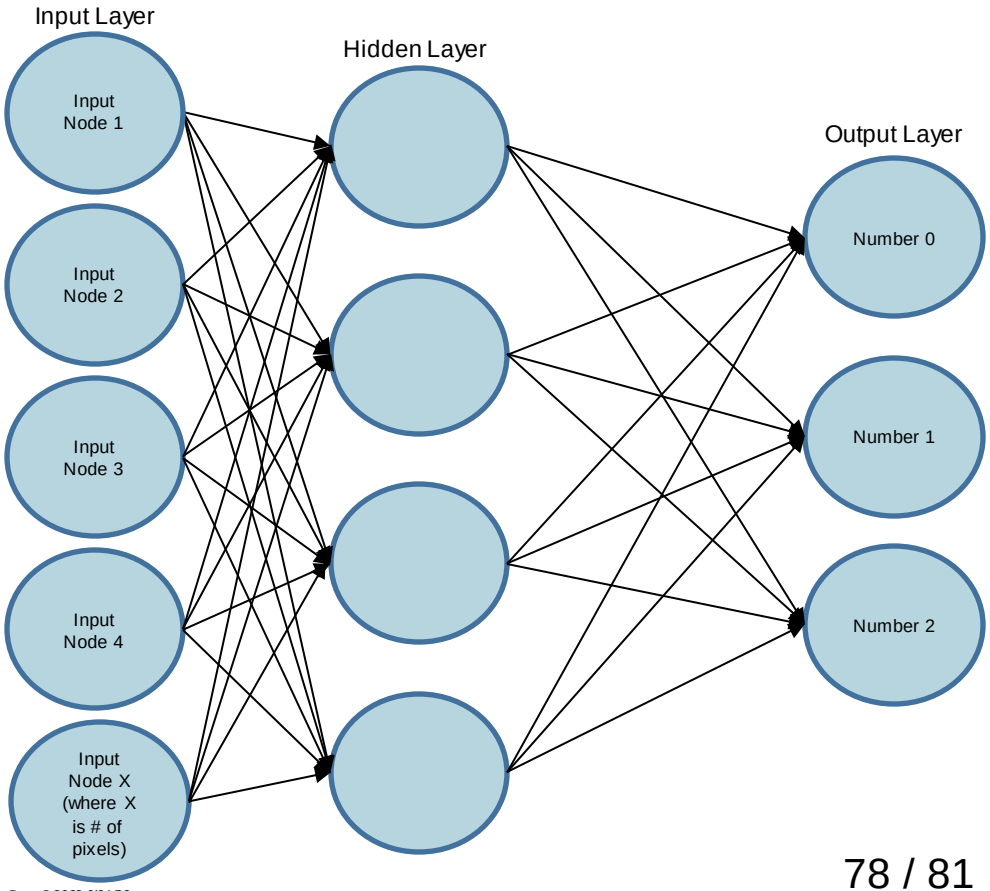
How Does a Feed Forward Neural Network Work? (2/2)

Step 2: Backwards Propagation

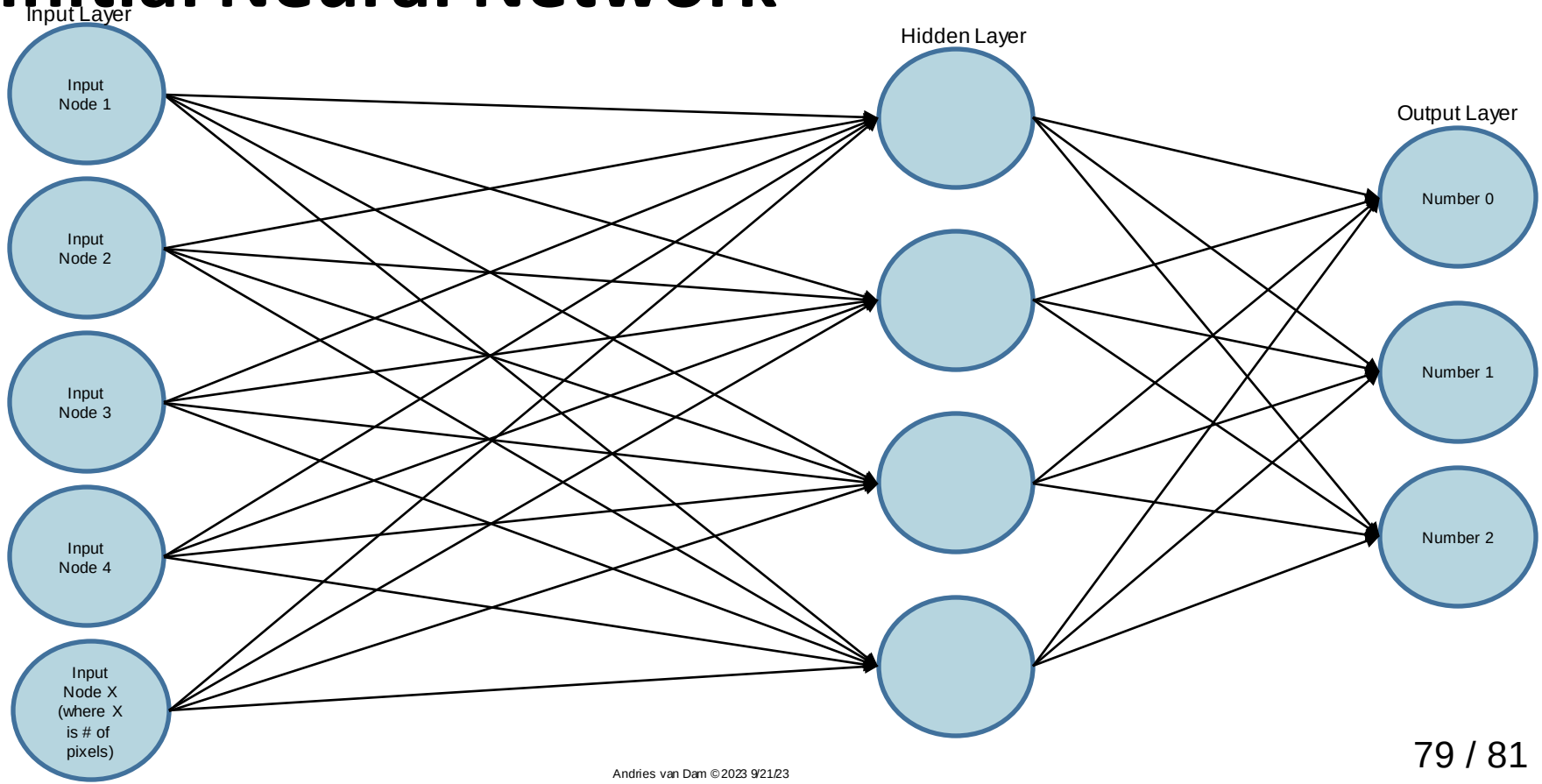
- **Calculate Loss**
  - Compute the loss (frequently Mean Square Error) of the predicted output vs. actual output
  - A measure of how much the actual output differs from the predicted output
- **Gradient Descent Algorithm**
  - Use calculus chain rule to work backwards and calculate which weights will minimize the loss (MSE) of the predicted output

Step 3: Repeat

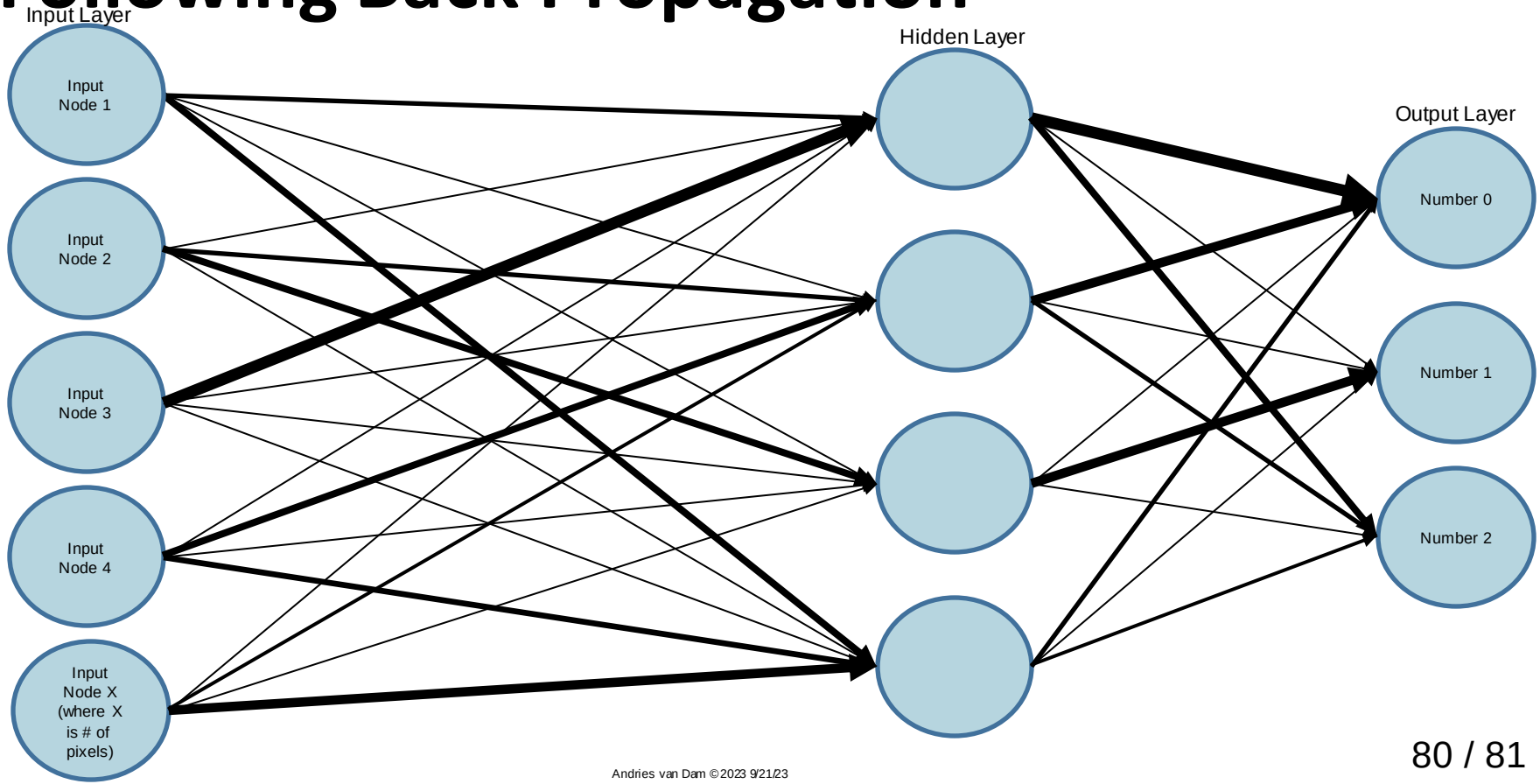
- Repeat the first two steps for either a set number of iterations or until loss drops below



Initial Neural Network



Following Back Propagation



Making the Leap to Generative AI?

Scale of LLMs:

- GPT-3 had **175 billion**

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Source.
1. Open AI
2. Semafor