INTERESTED IN CS RESEARCH?

CONTACT THE MURAS!

- Getting involved with lab opportunities
- Applying for UTRAs
- Grad school advice + peer mentorship
- Research symposium, open house

HOURS: FRIDAY 12-1PM EST
WEBSITE: UR.CS.BROWN.EDU

MURA@CS.BROWN.EDU
Lecture 4

Working with Objects: Variables, Containment, and Association
This Lecture:

- **Storing values in variables**
- **Methods that take in instances as parameters**
- **Containment and association relationships (how instances know about other instances in the same program)**
Review: Methods

- **Call methods**: give commands to an instance of a class
  
  ```java
  samBot.turnRight();
  ```

- **Define methods**: give a class specific capabilities
  
  ```java
  public void turnLeft() {
      // code to turn Robot left goes here
  }
  ```
Review: Constructors and Instances

- Declare a **constructor** (a method called whenever an instance is “born”)

  ```java
  public Calculator() {
    // code for setting up Calculator
  }
  ```

- Create an **instance** of a class with the `new` keyword

  ```java
  new Calculator();
  ```
Review: Parameters and Arguments

- **Define** methods that take in **parameters** (input) and have **return** values (output), e.g., this Calculator’s method:

  ```java
  public int add(int x, int y) {
    // x, y are dummy (symbolic) variables
    return (x + y);
  }
  ```

- **Call** such methods on instances of a class by providing **arguments** (actual values for symbolic parameters)

  ```java
  myCalculator.add(5, 8);
  ```
Review: Classes

- Recall that classes are just blueprints
- A class gives a basic definition of an object we want to model (one or more instances of that class)
- It tells the properties and capabilities of that object
- You can create any class you want and invent any methods and properties you choose for it!
Review: Instantiation

- **Instantiation** means building an instance from its class
  - a class can be considered a “blueprint,” where the capabilities of the instance are defined through the class’s methods
- Ex: `new Robot();` creates an instance of Robot by calling the Robot class’ **constructor** (see next slide)
Review: Constructors (1/2)

- A constructor is a method that is called to create a new instance
- Let’s define one for the Dog class
- Let’s also add methods for actions all Dogs know how to do like bark, eat, and wag their tails

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

    public void bark(int numTimes) {
        // code for barking goes here
    }

    public void eat() {
        // code for eating goes here
    }

    public void wagTail() {
        // code for wagging tail goes here
    }
}
```
Review: Constructors (2/2)

- Note constructors do not specify a return type
- Name of constructor must exactly match name of class
- Now we can instantiate a Dog in some method:

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

    public void bark(int numTimes) {
        // code for barking goes here
    }

    public void eat() {
        // code for eating goes here
    }

    public void wagTail() {
        // code for wagging tail goes here
    }
}
```

```
new Dog();
```
Variables

- Once we create a **Dog** instance, we want to be able to give it commands by calling methods on it!
- To do this, we need to name our **Dog**
- Can name an instance by storing it in a **variable**

```
Dog django = new Dog();
/* named after Django Reinhardt – see https://www.youtube.com/watch?v=plpSfvdCH0Q*/
```

- In this case, **django** is the variable, and it stores a newly created instance of **Dog**
  - the variable name **django** is also known as an “identifier”
- Now we can call methods on **django**, a specific instance of **Dog**
  - i.e. `django.wagTail();`
Syntax: Variable Declaration and Assignment

- To **declare** and **assign** a variable, thereby initializing it, in a single statement is:

  ```java
  Dog django = new Dog();
  ```

  - declaration
  - Instantiation, followed by assignment

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

- The “=” operator **assigns** the instance of `Dog` that we created to the variable `django`. We say “`django` gets a new `Dog`”

- Note: type of `value` must match declared `type` on left

- We can reassign as many times as we like (example soon)
Assignment vs. Equality

In Java:

\[\text{price} = \text{price} + 1;\]

• Means “add 1 to the current value of price and assign that to price.” We shorthand this to “increment price by 1”

In Algebra:

• \[\text{price} = \text{price} + 1\] is a logical contradiction
Values vs. References

- A variable stores information as either:
  - a value of a **primitive** (aka **base**) type (like `int` or `float`)
  - a reference to an instance (like an instance of `Dog`) of an arbitrary type stored elsewhere in memory
    - we symbolize a reference with an arrow

- Think of the variable like a box; storing a value or reference is like putting something into the box

- Primitives have a predictable memory size, while arbitrary instances of classes vary in size. Thus, Java simplifies its memory management by having a fixed size reference to an instance elsewhere in memory
  - “one level of indirection”

```java
int favNumber = 9;
Dog django = new Dog();
```

(somewhere else in memory)
Lecture Question

Given this code, fill in the blanks:

```java
int x = 5;
Calculator myCalc = new Calculator();
```

Variable x stores a __________, and myCalc stores a __________.

A. value, value
B. value, reference
C. reference, value
D. reference, reference
Example: Instantiation (1/2)

public class PetShop {

    public PetShop() {
        this.testDjango();
    }

    public void testDjango() {
        Dog django = new Dog();
        django.bark(5);
        django.eat();
        django.wagTail();
    }
}

- Let’s define a new class PetShop which has a testDjango() method.
  - don’t worry if the example seems a bit contrived…
- Whenever someone instantiates a PetShop, its constructor is called, which calls testDjango(), which in turn instantiates a Dog
- Then testDjango() tells the Dog to bark, eat, and wag its tail (see definition of Dog)
Another Example: Instantiation (2/2)

- Another example: can instantiate a MathStudent and then call that instance to perform a simple, fixed, calculation

- First, create new Calculator and store its reference in variable named myCalc

- Next, tell myCalc to add 2 to 6 and store result in variable named answer

- Finally, use System.out.println to print value of answer to the console!
Instances as Parameters (1/3)

- Methods can take in not just numbers but also instances as parameters.
- The `DogGroomer` class has a method `trimFur()`.
- `trimFur` method needs to know which `Dog` instance to trim the fur of.
- Method calling `trimFur` will have to supply a specific instance of a `Dog`, called `shaggyDog` in `trimFur`.
- Analogous to `void moveForward(int numberOfSteps);`.

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

    public void trimFur(Dog shaggyDog) {
        // code that trims the fur of shaggyDog
    }
}
```
Instances as Parameters (2/3)

- Where to call the `DogGroomer`'s `trimFur` method?
- Do this in the `PetShop` method `testGroomer()`.
- `PetShop`'s call to `testGroomer()` instantiates a `Dog` and a `DogGroomer`, then calls the `DogGroomer` to `trimFur` of the `Dog`.
- First two lines could be in either order.

```java
public class PetShop {
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.trimFur(django);
    }
}
```
Instances as Parameters (3/3): Flow of Control

0. In App’s constructor, a PetShop is instantiated (thereby calling PetShop’s constructor). Then:

1. The PetShop in turn calls the testGroomer() helper method, which instantiates a Dog and stores a reference to it in the variable django

2. Next, it instantiates a DogGroomer and stores a reference to it in the variable groomer

3. The trimFur method is called on groomer, passing in django as an argument; the groomer will think of it as shaggyDog, a synonym

```java
public class App {
    public App() {
        new Petshop();
    }
}

public class PetShop {
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.trimFur(django);
        //exit method, django and groomer disappear
    }
}
```
What is Memory?

- Memory ("system memory" aka RAM, not disk or other peripheral devices) is the hardware in which computers store information during computation.

- Think of memory as a list of slots; each slot holds information (e.g., an `int` variable, or a reference to an instance of a class).

- Here, two references are stored in memory: one to a `Dog` instance, and one to a `DogGroomer` instance.

```java
public class PetShop {
    public PetShop() {
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.trimFur(django);
    }
}
```
Instances as Parameters: Under the Hood (1/6)

Note: Recall that in Java, each class is stored in its own file. Thus, when creating a program with multiple classes, the program will work as long as all classes are written before the program is run. Order doesn't matter.
Instances as Parameters: Under the Hood (2/6)

When we instantiate a Dog, he’s stored somewhere in memory. Our PetShop will use the name django to refer to this particular Dog, at this particular location in memory.
Instances as Parameters: Under the Hood (3/6)

```java
public class PetShop {

    public PetShop() {
        this.testGroomer();
    }

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

```java
public class DogGroomer {

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

    public void trimFur(Dog shaggyDog) {
        // code that trims the fur of shaggyDog goes here!
    }
}
```

Somewhere in memory...

... Usually not adjacent in memory!

Same goes for the DogGroomer—we store a particular DogGroomer somewhere in memory. Our PetShop knows this DogGroomer by the name groomer.
Instances as Parameters: Under the Hood (4/6)

We call the `trimFur` method on our `DogGroomer`, `groomer`. We need to tell her which `Dog` to `trimFur` (since the `trimFur` method takes in a parameter of type `Dog`). We tell her to trim `django`.
Instances as Parameters: Under the Hood (5/6)

When we pass in `django` as an argument to the `trimFur` method, we’re telling the `trimFur` method about him. When `trimFur` executes, it sees that it has been passed that particular `Dog`. Somewhere in memory...
Instances as Parameters: Under the Hood (6/6)

The `trimFur` method doesn’t really care which `Dog` it’s told to trimFur—no matter what another instance’s name for the `Dog` is, `trimFur` is going to know it by the name `shaggyDog`.

```java
public class PetShop {

    public PetShop() {
        this.testGroomer();
    }

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

```java
public class DogGroomer {

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

    public void trimFur(Dog shaggyDog) {
        // code that trims the fur of shaggyDog goes here!
    }
}
```
Variable Reassignment (1/3)

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

- What if we wanted our `DogGroomer` to **trimFur** two different `Dog` instances when the `PetShop` opened?

- Could create another variable, or re-use the variable `django` to first point to one `Dog`, then another!

```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);
    }
}
```
Variable Reassignment (2/3)

- First, instantiate another Dog, and **reassign** variable `django` to point to it

- Now `django` no longer refers to the first Dog instance we created, which was already groomed

- Then tell groomer to `trimFur` the newer Dog. It will also be known as `shaggyDog` inside the `trimFur` method

```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(); // reassign django
        groomer.trimFur(django);
    }
}
```
Variable Reassignment (3/3)

- When we **reassign** a variable, we do not declare its type again, Java remembers from first time
- Can **reassign** to a brand new instance (like in PetShop) or to an already existing instance by using its identifier

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

- Now **django** and **scooby** refer to the same **Dog**, specifically the one that was originally **scooby**
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);
    }
}
Variable Reassignment: Under the Hood (2/5)

```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);
    }
}
```
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);
    }
}
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(); //old ref garbage collected - stay tuned!
        groomer.trimFur(django);
    }
}
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(); //old ref garbage collected – stay tuned!
        groomer.trimFur(django);
    }
}
Local Variables (1/2)

- All variables we’ve seen so far have been **local variables**: variables declared inside a method.

- Problem: the **scope** of a local variable (where it is known and can be accessed) is limited to its own method—it cannot be accessed from anywhere else.
  - same is true of method’s parameters

```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 (2/2)

- We created `groomer` and `django` in our `PetShop`'s helper method, but as far as the rest of the class is concerned, they don’t exist.
- Once the method is executed, they’re gone:
  - this is known as “Garbage Collection”

```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);
    }
}
```
“Garbage Collection”

- If an instance referred to by a variable goes out of scope, we can no longer access it. Because we can’t access the instance, it gets garbage collected
  - in garbage collection, the space that the instance took up in memory is freed and the instance no longer exists

- Lose access to an instance when:
  - local variables go out of scope at the end of method execution
  - variables lose their reference to an instance during variable reassignment (django, slide 35)
Accessing Local Variables

- If you try to access a local variable outside of its method, you’ll receive a “cannot find symbol” compilation error.

```java
public class PetShop {
    /* This is the constructor! */
    public PetShop() {
        DogGroomer groomer = new DogGroomer();
        this.cleanShop();
    }

    public void cleanShop() {
        //assume we've added a sweep method
        //to DogGroomer
        groomer.sweep();
        //other methods to empty trash, etc.
    }
}
```

In Terminal:

```
Petshop.java:13: error: cannot find symbol
  groomer.sweep();
  ^
symbol: variable groomer
location: class PetShop
```

scope of groomer
Introducing… Instance Variables!

● Local variables aren’t always what we want. We’d like every PetShop to come with a DogGroomer who exists for as long as the PetShop exists.

● That way, as long as the PetShop is in business, we’ll have our DogGroomer on hand.

● We accomplish this by storing the DogGroomer in an instance variable.

● It may seem unnatural to have a PetShop contain a DogGroomer, but it works in the kind of modeling that OOP makes possible – stay tuned.
What’s an Instance Variable?

- An **instance variable** models a property that all instances of a class have
  - its **value** can differ from instance to instance
- Instance variables are declared within a class, not within a single method, and are accessible from anywhere within the class – their **scope** is the entire class
- Instance variables and local variables are identical in terms of what they can store—either can store a base type (like an **int**) or a reference to an instance of some other class
Modeling Properties with Instance Variables (1/2)

- Methods model **capabilities** of a class (e.g., move, dance)
- All instances of same class have exact same methods (capabilities) **and the same properties**
- BUT: the potentially differing **values** of those **properties** can differentiate a given instance from other instances of the same class
- We use instance variables to model these properties and their values (e.g., the robot’s size, position, orientation, color, …)
Modeling Properties with Instance Variables (1/2)

- All instances of a class have same set of properties, but values of these properties will differ
- E.g. CS15Students might have property “height”
  - for one student, the value of “height” is 5’2”. For another, it’s 6’4”
- CS15Student class would have an instance variable to represent height
  - value stored in this instance variable would differ from instance to instance
When should I define an instance variable?

- In general, variables that fall into one of these three categories should be instance variables of the class rather than local variables within a method:
  - **attributes**: simple descriptors of an instance, e.g., color, height, age, ...; the next two categories encode relationships between instances
  - **components**: “parts” that make up an instance. If you are modeling a car, the car’s engine and doors will be used in multiple methods, so they should be instance variables; ditto PetShop and its DogGroomer
  - **associations**: a relationship between two instances in which one instance knows about the other, but they are not necessarily part of each other. For example, the instructor needs to know about TAs (more on this soon), but the instructor is not a part of the TA class – they are peers.

- All methods in a class can access all its properties, to use them and/or change them.
Instance Variables (1/4)

- We’ve modified PetShop example to make our DogGroomer an instance variable for the benefit of multiple methods – yes, DogGroomer here is considered a component (part) of the PetShop.

- Split up declaration and assignment of instance variable:
  1. **declare** instance variable at the top of the class, to notify Java compiler.
  2. **initialize** the instance variable by assigning a value to it in the constructor.
  3. Primary purpose of constructor is to initialize all instance variables so the instance has a valid initial “state” at its “birth”; it typically should do no other work.
  4. **State** is the set of all values for all properties—local variables don’t hold properties; they are “temporaries”.

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

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
Instance Variables (2/4)

- Note we include the keyword `private` in declaration of our instance variable
- `private` is an access modifier, just like `public`, which we’ve been using in our method declarations

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

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
Instance Variables (3/4)

- If declared as `private`, the method or instance variable can only be accessed inside the class – their scope is the entire class.

- If declared as `public`, can be accessed from anywhere – their scope can include multiple classes.

- In CS15, you’ll declare instance variables as `private`, with rare exception!

- Note that local variables don’t have access modifiers -- they always have the same scope (their own method).

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

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
Instance Variables (4/4)

- CS15 instance variable rules:
  - start instance variable names with an **underscore** to easily distinguish them from local variables
  - make all instance variables **private** so they can only be accessed from within their own class!
  - **encapsulation** for safety…your properties are your private business. We will also show you safe ways of allowing other classes to have selective access to designated properties… stay tuned.

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

    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
Always Remember to Initialize!

- What if you declare an instance variable, but forget to initialize it? What if you don’t supply a constructor and your instance variables are not initialized?

- The instance variable will assume a “default value”
  - if it’s an `int`, it will be 0
  - if it’s an instance, it will be `null`—a special value that means your variable is not referencing any instance at the moment

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

    /* This is the constructor! */
    public PetShop() {
        // oops! Forgot to initialize _groomer
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
NullPointerExceptions

- If a variable’s value is null and you try to give it a command, you’ll be rewarded with a *runtime error*—you can’t call a method on “nothing”!

- `_groomer`’s default value is null so this particular error yields a NullPointerException

- When you run into one of these (we promise, you will), make sure all variables have been explicitly initialized, preferably in the constructor, and none are initialized as null

```java
class PetShop {
    private DogGroomer _groomer;

    public PetShop() {
        // oops! Forgot to initialize _groomer
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```
Instance Variables Example (1/2)

- Let’s add an instance variable to the Dog class
- _furLength stores an int that keeps track of the length of a Dog’s fur
- _furLength is assigned a default initial value of 3 in the constructor – it can be changed later, of course

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

    public Dog() {
        _furLength = 3;
    }

    /* bark, eat, and wagTail elided */
}
```
Instance Variables Example (2/2)

- \_furLength is a **private** instance variable—only accessible within Dog class
- What if another instance needs to know or change the value of \_furLength?
- When a DogGroomer trims the fur of a Dog, it needs to update \_furLength

```java
class Dog {
    private int \_furLength;
    public Dog() {
        \_furLength = 3; /* all dogs have the same furLength initially */
    }
    /* bark, eat, and wagTail elided */
}
```
Accessors / Mutators (1/3)

- A class may make the value of an instance variable publicly available via an **accessor method** that **returns** the value when called.

- `getFurLength` is an accessor method for `_furLength`.

- Can call `getFurLength` on an instance of `Dog` to **return** its current `_furLength` value.

- Remember: return type specified and value returned must match!

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

    public Dog() {
        _furLength = 3;
    }

    public int getFurLength() {
        return _furLength;
    }
}
/* bark, eat, and wagTail elided */
```

return type is `int`

value returned, type `int`
Accessors / Mutators (2/3)

- Similarly, a class may define a **mutator method** which allows another class to change the value of some instance variable.
- `setFurLength` is a mutator method for `_furLength`.
- Another instance can call `setFurLength` on a `Dog` to change the value stored in `_furLength`.

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

    public Dog() {
        _furLength = 3;
    }

    public int getFurLength() {
        return _furLength;
    }

    public void setFurLength(int furLength) {
        _furLength = furLength;
    }

    /* bark, eat, and wagTail elided */
}
```
Accessors / Mutators (3/3)

- Fill in DogGroomer’s `trimFur` method to modify `furLength` of the Dog it is trimming the fur of.

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

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

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Example: Accessors (1/2)

Check that the trimFur method 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() {
        _groomer = new DogGroomer();
        this.testGroomer();
    }

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

We use the accessor getFurLength to retrieve the value django stores in its _furLength instance variable
Example: Accessors (2/2)

- What values print out to the console?

public class PetShop {
    private DogGroomer _groomer;

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

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

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

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

- first, 3 is printed because 3 is the initial value we assigned to _furLength in the Dog constructor (slide 54)
- next, 1 prints out because groomer just set django’s _furLength to 1
Example: Mutators

- What if we don’t always want to trim the 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 {
    public void testGroomer() {
        Dog django = new Dog();
        _groomer.trimFur(django, 2);
    }
}
```

```java
public class DogGroomer {
    public void trimFur(Dog shaggyDog, int furLength) {
        shaggyDog.setFurLength(furLength);
    }
}
```

- `groom` 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 parameters 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!
Summary of Accessors/Mutators

- Instance variables should always be declared `private` for safety, and should be declared at the top of class definition
  - but classes may want to offer useful functionality that allows access to selective properties (instance variables).

- If we made such instance variables `public`, any method could change them, i.e., with the caller in control of the inquiry or change – this is totally 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.
Containment and Association

- **Key to OOP:** how are different classes related to each other so their instances can communicate to collaborate?

- Relationships established via containment or association

- Often a class A will need as a component an instance of class B, stored in an instance variable. A will create the instance of B by using the `new` keyword. We say A contains that instance of class B. Thus A knows about B and can call methods on it. Note this is **not symmetrical**: B can’t call methods on A!
  
  - thus a car can call methods of a contained engine but the engine can’t call methods on the car

- At other times, a class C will need to “know about” an instance of class D, where the instance of class D is not created by class C. An instance of class D is passed into the constructor of class C as an argument. We say that C and D are associated with each other. This is also non-symmetric: D doesn’t automatically know about C.
  
  - can make association symmetric by separately telling D to be associated with C

- This is all very abstract… Let’s see code!
Example: Containment

- PetShop contains a DogGroomer instance

- Containment relationship because PetShop itself instantiates a DogGroomer instance \_groomer with "new DogGroomer();"

- Since PetShop created a DogGroomer and stored it in an instance variable, all PetShop’s methods “know” about the \_groomer and can access it

```java
public class PetShop {

    private DogGroomer _groomer;

    public PetShop() { //constructor
        _groomer = new DogGroomer();
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); //local var
        _groomer.trimFur(django);
    }

    // other methods...
}
```
Association (1/8)

- Now let’s set up an association!

- **Association** means an instance of one class “knows about” an instance of another class that is not one of its components

```java
public class DogGroomer {

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

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Motivation for Association (2/8)

- As noted, *PetShop* contains a *DogGroomer*, so it can send messages to the *DogGroomer*

- But what if the *DogGroomer* needs to send messages to the *PetShop* she works in?
  - The *DogGroomer* probably needs to know several things about her *PetShop*: for example, operating hours, grooming supplies in stock, customers currently in the shop...

```java
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Association (3/8)

- The PetShop keeps track of such information in its properties (not shown here)
- We can set up an association so DogGroomer can send her PetShop messages to retrieve information from it as needed

```java
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Example: Setting up the Association (4/8)

- To set up the association, we must modify `DogGroomer` to store the knowledge of the `_petShop`
- To set it up, declare an instance variable named `_petShop` in the `DogGroomer`
- But how to initialize this instance variable? Such initialization should be done in `DogGroomer`'s constructor

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

    public DogGroomer() {
        _petShop = ???
    }

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Example: Setting up the Association (5/8)

- We modify `DogGroomer`'s constructor to take in a parameter of type `PetShop`.
- Constructor will refer to it by the name `myPetShop`. To “remember” the passed argument, the constructor stores it in the `_petShop` instance variable.

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

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the assoc.
    }
    //trimFur method elided
}

public class PetShop {
    private DogGroomer _groomer;

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

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
}
```

Code from previous slides

- We modify `DogGroomer`'s constructor to take in a parameter of type `PetShop`.
- Constructor will refer to it by the name `myPetShop`. To “remember” the passed argument, the constructor stores it in the `_petShop` instance variable.
Example: Setting up the Association (6/8)

- What argument should DogGroomer's constructor store in _petShop?
  - The PetShop instance that created the DogGroomer

- How?
  - By passing this as the argument
    - i.e., the PetShop tells the DogGroomer about itself

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

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop;  // store the assoc.
    }
    //trimFur method elided
}
```

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

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

    public void testGroomer() {
        Dog django = new Dog(); // local var
        _groomer.trimFur(django);
    }
```
Example: Setting up the Association (7/8)

- Now, the instance variable, `_petShop`, records the instance of `PetShop`, called `myPetShop`, that the `DogGroomer` belongs to

- `_petShop` now points to same `PetShop` instance passed to its constructor

- After constructor has been executed and can no longer reference `myPetShop`, any `DogGroomer` method can still access same `PetShop` instance by the name `_petShop`

```
public class DogGroomer {

    private PetShop _petShop;

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the assoc.
    }

    public void trimFur(Dog shaggyDog) {
        shaggyDog.setFurLength(1);
    }
}
```
Example: Using the Association (8/8)

- Let’s say we’ve written an **accessor** method and a **mutator** method in the `PetShop` class:
  - `getClosingTime()` and `setNumCustomers(int customers)`

- If the `DogGroomer` ever needs to know the closing time, or needs to update the number of customers, she can do so by calling
  - `getClosingTime()`
  - `setNumCustomers(int customers)`

```java
public class DogGroomer {
    private PetShop _petShop;
    private Time _closingTime;

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store assoc.
        _closingTime = _petShop.getClosingTime();
        _petShop.setNumCustomers(20);
    }
}
```
public class PetShop {
    private DogGroomer _groomer;

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

    public void testGroomer() {
        Dog django = new Dog();
        _groomer.trimFur(django);
    }
}

public class DogGroomer {
    private PetShop _petShop;

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop;
    }

    /* trimFur and other methods elided for this example */
}
public class PetShop {
    private DogGroomer _groomer;

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

    public void testGroomer() {
        Dog django = new Dog();
        _groomer.trimFur(django);
    }
}

public class DogGroomer {
    private PetShop _petShop;

    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop;
    }

    /* trimFur and other methods elided for this example */
}

Association: Under the Hood (2/5)

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 variables (just a DogGroomer here)
The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor (remember the this keyword?)
When the DogGroomer’s constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument by the caller, i.e., the PetShop itself.
The DogGroomer sets its `_petShop` instance variable to point to the same PetShop it received as an argument (see slide 68!). Now it “knows about” the PetShop that instantiated it, and so do all its methods...
Lecture 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 _teacher;
    public School() {
        _teacher = new Teacher(this);
    }
    //additional methods, some using
    //_teacher
}

public class Teacher {
    private School _school;
    public Teacher(School school) {
        _school = school;
    }
    //additional methods, some using
    //_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
Lecture Question Review

public class School{
    private Teacher _teacher;

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

public class Teacher{
    private School _school;

    public Teacher(School school) {
        _school = school;
    }
    //additional methods, some using
    //_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
Another Example: Association (1/6)

- Here we have the class `CS15Professor`
- We want `CS15Professor` to know about his Head TAs—he didn’t create them or vice versa, hence 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 */
}
```
Another Example: Association (2/6)

- 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 */
}
```
Another Example: Association (3/6)

- Here’s our solution!
- Remember, you can choose your own names for the instance variables and parameters
- The `CS15Professor` can now send a message to one of his `HeadTAs` like this:

  ```java
  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) {
          _hta1 = firstTA;
          _hta2 = secondTA;
          _hta3 = thirdTA;
          _hta4 = fourthTA;
          _hta5 = fifthTA;
      }

      /* additional methods elided */
  }
  
  _hta2.setUpLecture();
  ```
Another Example: Association (4/6)

- 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 `HeadTA` class takes no parameters in its constructor

```java
public class CS15App {
  // declare CS15Professor instance var
  // declare five HeadTA instance vars
  // ...
  // ...
  public CS15App() {
    // instantiate the professor!
    // ...
    // ...
    // instantiate the five HeadTAs
  }
}
```
Another Example: Association (5/6)

- We declare _andy, _aalia, _anna, _gil, _marina, and _will as instance variables
- In the constructor, we instantiate them
- Since the constructor of CS15Professor takes in 5 HeadTAs, we pass in _aalia, _anna, _gil, _marina, and _will

```java
class CS15App {
    private CS15Professor _andy;
    private HeadTA _aalia;
    private HeadTA _anna;
    private HeadTA _gil;
    private HeadTA _marina;
    private HeadTA _will;

    public CS15App() {
        _aalia = new HeadTA();
        _anna = new HeadTA();
        _gil = new HeadTA();
        _marina = new HeadTA();
        _will = new HeadTA();
        _andy = new CS15Professor(_aalia, _anna, _gil, _marina, _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) {
        _hta1 = firstTA;
        _hta2 = secondTA;
        _hta3 = thirdTA;
        _hta4 = fourthTA;
        _hta5 = fifthTA;
    }

    /* additional methods elided */
}

public class CS15App {
    private CS15Professor _andy;
    private HeadTA _aalia;
    private HeadTA _anna;
    private HeadTA _gil;
    private HeadTA _marina;
    private HeadTA _will;

    public CS15App() {
        _aalia = new HeadTA();
        _anna = new HeadTA();
        _gil = new HeadTA();
        _marina = new HeadTA();
        _will = new HeadTA();
        _andy = new CS15Professor(_aalia,
                                  _anna, _gil, _marina, _will);
    }
}
More Associations (1/5)

- 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 _andy as a parameter into each **HeadTAs** constructor?

```java
public class CS15App {
    private CS15Professor _andy;
    private HeadTA _aalia;
    private HeadTA _anna;
    private HeadTA _gil;
    private HeadTA _marina;
    private HeadTA _will;
    public CS15App() {
        _aalia = new HeadTA();
        _anna = new HeadTA();
        _gil = new HeadTA();
        _marina = new HeadTA();
        _will = new HeadTA();
        _andy = new CS15Professor(_aalia, _anna, _gil, _marina, _will);
    }
}
```
When we instantiate \_aalia, \_anna, \_gil, \_marina, and \_will, we would like to use a modified HeadTA constructor that takes an argument, \_andy.

But \_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
public class CS15App {
    private CS15Professor \_andy;
    private HeadTA \_aalia;
    private HeadTA \_anna;
    private HeadTA \_gil;
    private HeadTA \_marina;
    private HeadTA \_will;

    public CS15App() {
        \_aalia = new HeadTA();
        \_anna = new HeadTA();
        \_gil = new HeadTA();
        \_marina = new HeadTA();
        \_will = new HeadTA();
        \_andy = new CS15Professor(\_aalia, \_anna, \_gil, \_marina, \_will);
    }
}
```
More Associations (3/5)

- Instantiate _aalia, _anna, _gil, _marina, and _will before we instantiate _andy
- Use a new method (mutator), setProf, and pass _andy to each HeadTA

```java
public class CS15App {
    private CS15Professor _andy;
    private HeadTA _aalia;
    private HeadTA _anna;
    private HeadTA _gil;
    private HeadTA _marina;
    private HeadTA _will;

    public CS15App() {
        _aalia = new HeadTA();
        _anna = new HeadTA();
        _gil = new HeadTA();
        _marina = new HeadTA();
        _will = new HeadTA();
        _andy = new CS15Professor(_aalia,
                                _anna, _gil, _marina, _will);

        _aalia.setProf(_andy);
        _anna.setProf(_andy);
        _gil.setProf(_andy);
        _marina.setProf(_andy);
        _will.setProf(_andy);
    }
}
```
public class HeadTA {
    private CS15Professor _professor;

    public HeadTA() {
        //Other code elided
    }

    public void setProf(CS15Professor prof) {
        _professor = prof;
    }
}

• Now each HeadTA will know about _andy!

public class CS15App {
    private CS15Professor _andy;
    private HeadTA _aalia;
    private HeadTA _anna;
    private HeadTA _gil;
    private HeadTA _marina;
    private HeadTA _will;

    public CS15App() {
        _aalia = new HeadTA();
        _anna = new HeadTA();
        _gil = new HeadTA();
        _marina = new HeadTA();
        _will = new HeadTA();
        _andy = new CS15Professor(_aalia, _anna, _gil, _marina, _will);

        _aalia.setProf(_andy);
        _anna.setProf(_andy);
        _gil.setProf(_andy);
        _marina.setProf(_andy);
        _will.setProf(_andy);
    }
}
More Associations (5/5)

- 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`!
- So this is not a completely satisfactory solution, but we will learn more tools soon that will allow us to develop a more complete solution
Visualizing Containment and Association

CS15App

CS15Professor

HeadTA

“contains one instance of”

“contains more than one instance of”

“knows about”
Summary

Important concepts:

- Using **local variables**, whose scope is limited to a method
- Using **instance variables**, which store the properties of instances of a class for use by multiple methods—use them only for that purpose
- A variable that “goes out of scope” is **garbage collected**
  - for a local variable when the method ends
  - for an instance when the last reference to it is deleted
- **Containment**: when one instance is a component of another class so the container can therefore send messages to the component it created
- **Association**: when one class knows about an instance of a different class that is not one of its components—has to be set up explicitly
Announcements

● Lab1: Java Objects begins today!
  o If you have not received an email about your permanent section time please contact the HTAs ASAP
  o Check out the website for the pre-section work
    ▪ Pre-Lab video and video quiz (for before your section time)
    ▪ SRC Pre-Section Reading (one page with a lab activity preview!)

● AndyBot due Thursday 2/4 at 11:59 p.m. EST
  o No late hand in date! Make sure you submit AndyBot on time!
IT in the News

ft. Socially Responsible Computing!
Talk: Fairness and Bias in Algorithmic Decision-Making

Jon Kleinberg,
Cornell University

Wednesday February 3,
12:00pm - 1:00pm

More Info:
https://sites.google.com/view/seam-seminar/home

Zoom Link:
https://brown.zoom.us/j/91038690385

Abstract: As algorithms trained via machine learning are increasingly used as a component of screening decisions in areas such as hiring, lending, and education, discussion in the public sphere has turned to the question of what it means for algorithmic classification to be fair to different groups. We consider several of the key fairness conditions that lie at the heart of these debates, and discuss recent research on trade-offs and interventions through the lens of these conditions. We also explore how the complexity of a classification rule interacts with its fairness properties, showing how natural ways of approximating a classifier via a simpler rule can lead to unintended biases in the outcome.
Who Owns the News? (1/2)

- For years, Google has provided automatically-generated article previews in search results – without paying publishers

- January 21, 2021:
  - Google agrees to pay publishers in France for content linked via search
  - but refuses similar law in Australia: threatens to block search, retaliate against Australian media
    - Facebook backs Google in aggressive response

- Same day, opposite response: why?
  - **power:** French agreement lets Google set terms, Australian involves independent arbiter
  - **money:** threatens Google's business model
    - based on tracking clicks, adding ads to search results, collecting data (e.g., to profile and microtarget) and selling data to third parties

Melanie Silva, Managing Director of Google AU & NZ, appears at Australian Senate inquiry

*image source: NYT, Jan 21, 2021*
Who Owns the News? (2/2)

● What’s at stake?
  ○ future of news media & publishing
    ■ newspaper & magazine readership & revenue have dropped catastrophically, threatening journalism
      ○ “fake news” → serious change in trust of news publications
    ■ Google & Facebook previews discourage click-through, decreasing publisher revenue
  ○ free & open Internet (possibly)
    ■ dissenters to Australia law include Sir Tim Berners-Lee (WWW)

● Should Google be able to deny Search to an entire country?
  ○ what does this say about their power?
  ○ what does this say about your power (as CS students)?

● What responsibilities do governments have in regulating (or not) Big Tech?
  ○ Trump and Biden on same side?!? (re: Section 230 of Communications Decency Act of 1996)

● Next time: what responsibilities do platforms have for content appearing on their sites?

“What Google returns is more of a media-rich, detailed preview than a simple link….This can obviously decrease revenue for news providers, as well as perpetuate misinformation.”
  – Tama Leaver, professor of Internet Studies, Curtin University (Perth, AU)

“The ability to link freely -- meaning without limitations regarding the content of the linked site and without monetary fees -- is fundamental to how the web operates.”
  – Sir Tim Berners-Lee (founded WWW)