CS15 Cookie Mixer

- CIT 401 (Library) Wednesday, Sept. 23 4pm - 5pm
- Celebrate completing AndyBot at the CS15 Cookie Mixer this Wednesday!
- Strongly encourage women, students of color, LGBTQ students, and 15ers just starting out in CS to attend!
- Mingle with your classmates, upperclassmen & former 15ers. Learn more about the department and talk about the ways we can improve it!

Lecture 4
Working with Objects: Variables, Containment, and Association

This Lecture:
- Storing values in variables
- Methods that take in objects as parameters
- Containment and association relationships (how objects know about other objects in the same program)
- Packages (collections of related classes) and how to import classes from other packages and use them in your code

What We Already Know How to Do
- Call methods: send messages to an object
  ```java
  andyBot.turnRight();
  ```
- Define methods: give a class specific capabilities
  ```java
  public void turnLeft() {
      // code to turn Robot left goes here
  }
  ```
What We Already Know How to Do

- Declare a constructor (a method called whenever an object is “born”)
  
  ```java
  public Calculator() {
    // code for setting up Calculator
  }
  ```

- Create an instance of a class with the new keyword
  
  ```java
  new Calculator();
  ```

What We Already Know How to Do

- Write methods that take in parameters (input) and have return values (output)
  
  ```java
  public int add(int x, int y) {
    return x + y;
  }
  ```

- Call such methods by providing arguments (actual values for symbolic parameters)
  
  ```java
  myCalculator.add(5, 8);
  ```

Review: Classes

- As we’ve mentioned, classes are just blueprints
- A class gives us a basic definition of something we want to model
- It tells us what the properties and capabilities of that kind of thing are (we’ll deal with properties in this lecture)
- Can create a class called pretty much anything you want, and invent any methods and properties you choose for it!

Review: Instantiation

- Instantiation means creating an object from its class “blueprint”
- Ex: new Robot(); creates an instance of Robot
- This calls the Robot class’s constructor: a special kind of method
Review: Constructors

- A constructor is a method that is called to create a new object
- Let’s define one for the Dog class
- All Dogs know how to 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

- 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
new Dog();
```

Variables

- Once we create a Dog, we want to be able to send it messages by calling methods on it!
- To do this, we need to name our Dog
- Can name an object by storing it in a variable, like this:

```java
Dog sam = new Dog();
```
- A variable stores a value
- In this case, sam is the variable, and it stores a newly created instance of Dog

Syntax: Variable Declaration and Assignment

- Typical form for declaring and assigning a variable is:

```java
<type> <name> = <value>;
```
- Note: type of value must match the declared type on the left
- If you want, you can also split the declaration and assignment up into two steps:

```java
<type> <name>;  // declaration
<name> = <value>;  // assignment
```
Variables

Dog sam = new Dog();

- The “=” operator assigns the instance of Dog that we created to the variable sam. We say “sam gets a new Dog”
- Now we can call methods on our Dog using its new name (sam)

Variables: Values vs. References

- A variable stores either:
  - a value of a primitive type (like int or float)
  - or a reference to an instance of Dog stored elsewhere in memory for an instance of an arbitrary type

Dog sam = new Dog();

- Think of the variable like a box
- Primitives have a predictable memory size, while arbitrary objects vary in size, hence the need for a reference

Instantiation in Context

public class PetShop {
    public PetShop() {
        Dog sam = new Dog();
        sam.bark(5);
        sam.eat();
        sam.wagTail();
    }
}

- Let’s instantiate sam the Dog within the constructor of the PetShop class
- Whenever someone instantiates a PetShop, it in turn instantiates a Dog
- Then it tells the Dog to bark, eat, and wag its tail

public class MathStudent {
    public void performCalculations() {
        Calculator myCalc = new Calculator();
        int answer = myCalc.add(2, 6);
        System.out.println(answer);
    }
}

- Another example: a MathStudent instantiates a Calculator
- First, create a new Calculator and store it 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
Objects as Parameters

- Methods can take in objects as parameters
- The DogGroomer class has a method groom
- groom method needs to know which Dog to groom

```java
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog
    }
}
```

Objects as Parameters

- The DogGroomer's groom method takes in a single parameter—a Dog
- Always specify type, then name of parameter
- Here, Dog is the type and "shaggyDog" is the name we've chosen

```java
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog
    }
}
```

Objects as Parameters

- How to call the groom method?
- Do this in the PetShop class
- PetShop instantiates a Dog and a DogGroomer, then tells the DogGroomer to groom the Dog

```java
public class PetShop {
    /* This is the constructor */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
    }
}
```

Objects as Parameters

- Elsewhere in the program, some method instantiates a PetShop (by calling PetShop's constructor). Then:
  1. The PetShop return instantiates a Dog and stores a reference to it in the variable dogE
  2. Next, it instantiates a DogGroomer and stores a reference to it in the variable groomer
  3. The groom method is called on groomer, passing in dogE as an argument

```java
public class PetShop {
    /* This is the constructor */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
    }
}
Objects as Parameters: Under the Hood

public class PetShop {
    /* This is the constructor! */
    public PetShop() {
        Dog dog = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dog);
    }
}

public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
    }
}

Somewhere in memory ...

When we instantiate a Dog, he's stored somewhere in memory. Our PetShop will use the name dog to refer to this particular Dog at this particular location in memory.

We call the groom method on our DogGroomer, groomer. We need to tell her which Dog to groom (since the groom method takes a parameter of type Dog). We tell her to groom dog.

The same goes for the DogGroomer—we store a particular DogGroomer somewhere in memory. Our PetShop knows this Groomer by the name groomer.
Objects as Parameters: Under the Hood

```java
public class PetShop {
    /* This is the constructor */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
    }
}
```

```
public class DogGroomer {
    public DogGroomer() {
        // this is the constructor!
    }
    public void groom(Dog shaggyDog) {
        // code that grooms shaggyDog goes here!
    }
}
```

Somewhere in memory...

When we pass `dogE` as an argument to the `groom` method, we're telling the `groom` method about him. When `groom` executes, it sees that it has been passed that particular `Dog`.

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

Variable Reassignment

- After giving a variable an initial value, we can reassign it (make it refer to a different object)
- What if we wanted our `DogGroomer` to groom two different `Dogs` when the `PetShop` opened?
- Could we re-use the variable `dogE` to first point to one `Dog`, then another?

```java
public class PetShop {
    /* This is the constructor */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
        dogE = new Dog(); // reassign dogE
        groomer.groom(dogE);
    }
}
```

- First, instantiate another `Dog`, and reassign variable `dogE` to point to it
- This means that `dogE` no longer refers to the first `Dog` we created, which has already been groomed
- We then tell `groomer` to groom the newer `Dog`
public class PetShop {
    /* This is the constructor! */
    public PetShop() {
        Dog dog1 = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dog1);
        Dog dog2 = new Dog();
        groomer.groom(dog2);
    }
}
Variable Reassignment: Under the Hood

```java
public class PetShop {
    // This is the constructor!
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
        dogE = new Dog();
        groomer.groom(dogE);
    }
}
```

Local Variables

- All variables we've seen so far have been local variables: variables declared within a method
- Problem: the scope of a local variable is limited to its own method—it cannot be accessed from anywhere else
  - the same is true of method parameters

Local Variables

- We created `groomer` and `dogE` in our `PetShop`'s constructor, but as far as the rest of the class is concerned, they don't exist
- Once the constructor is executed, they're gone:

```
public class PetShop {
    /* This is the constructor!
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.groom(dogE);
        dogE = new Dog();
        groomer.groom(dogE);
    }
```

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 can accomplish this by storing the `DogGroomer` in an instance variable
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, but not within a single method, and are accessible from anywhere within the 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 object (instance of some other class)

Modeling Properties with Instance Variables

- Methods model the capabilities of a class
- 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

- All instances of a class have the same properties, but the values of these properties will differ
  - All CS15Student might have property “height”
    - For one student, the value of “height” is 5’2”. For another, it’s 6’4”
  - The 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 rather than local variables:
  - attributes: descriptors of an object, e.g., color, height, age, …
  - components: “parts” of an object. If you are modeling a car, its engine and doors might be instance variables
  - associations: things that are not part of an object, but that the object needs to know about. For example, Andy needs to know about his CS15 TAs (more on this soon)
- All methods in a class can access all its properties, to use them and/or to change them
Instance Variables

- We've modified PetShop example to make our DogGroomer an instance variable
- Split up declaration and assignment of instance variable:
  1. Declare instance variable at the top of the class, to not let Java initialise the instance variable by assigning a value
  2. Purpose of constructor is to initialise all instance variables in the instance has a valid initial "state" at its "birth"
  3. State is the set of all values for all properties—we don't consider local variables to be properties; they are "temporaries"

```java
class PetShop {
    private DogGroomer _groomer;
    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        Dog dog = new Dog();
        _groomer.groom(dog);
    }
}
```

- Note that 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
class PetShop {
    private DogGroomer _groomer;
    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer();
        Dog dog = new Dog();
        _groomer.groom(dog);
    }
}
```

Instance Variables

- If declared as private, the method or instance variable can only be accessed inside the class
- If declared as public, can be accessed from anywhere
- In CS15, you'll primarily declare instance variables as private and methods as public
- Note that local variables don't have access modifier—they always have the same scope (their own method)

```java
class PetShop {  
    private DogGroomer _groomer;  
    /* This is the constructor! */  
    public PetShop() {  
        _groomer = new DogGroomer();  
        Dog dog = new Dog();  
        _groomer.groom(dog);  
    }  
}
```

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, and you publish only those properties you want others to have access to

```java
class PetShop {  
    private DogGroomer _groomer;  
    /* This is the constructor! */  
    public PetShop() {  
        _groomer = new DogGroomer();  
        Dog dog = new Dog();  
        _groomer.groom(dog);  
    }  
}
```
Always Remember to Initialize!

- What if you declare an instance variable, but forget to initialize it?
- The instance variable will assume a "default value"
  - if it's an int, it will be 0
  - if it's an object, it will be null—
    a special value that means your
    variable is not referencing any
    instance at the moment

public class PetShop {
    private DogGroomer _groomer;
    /* This is the constructor */
    public PetShop(){
        Dog dog = new Dog();
        // Oops!
        _groomer.groom(dog);
    }
}

NullPointe rExceptions

- If a variable's value is null and
  you try to send it a message,
  you'll be rewarded with a runtime
  error—you can't call a method
  on "nothing"
- This particular error yields a
  NullPointerException
- When you run into one of these
  (we promise, you will)—edit
  your
  program to make sure you have
  explicitly initialized all variables

public class PetShop {
    private DogGroomer _groomer;
    /* This is the constructor */
    public PetShop(){
        Dog dog = new Dog();
        // Oops!
        _groomer.groom(dog);
    }
}

Instance Variables

- Let's add an instance variable to the Dog class
- _hairLength is an int that
  will keep track of length of a
  Dog's hair
- _hairLength is assigned a
  default value of 3 in the
  constructor

public class Dog {
    private int _hairLength;
    public Dog(){
        _hairLength = 3;
    }

    /* bark, eat, and wagTailelided */
}

Instance Variables

- _hairLength is a private
  instance variable—can only
  be accessed from within Dog
  class
- What if another object needs
  to know or change the value
  of _hairLength?
- When a DogGroomer grooms
  a Dog, it needs to update
  _hairLength

public class Dog {
    private int _hairLength;
    public Dog(){
        _hairLength = 3;
    }

    /* bark, eat, and wagTailelided */
}
Accessors/Mutators

- The class may make the value of an instance variable available via an accessor method
- `getHairLength` is an accessor method for `_hairLength`
- Another object can call `getHairLength` on an instance of `Dog` to retrieve its current `_hairLength` value

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

    public Dog() {
        _hairLength = 3;
    }

    public int getHairLength() {
        return _hairLength;
    }

    /* bark, eat, and wagTIlle */
}
```

Accessors/Mutators

- Similarly, a class may provide a mutator method to allow another class to change the value of one of its instance variables
- `setHairLength` is a mutator method for `_hairLength`
- Another object can call `setHairLength` on a `Dog` to change the value it stores in `_hairLength`

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

    public Dog() {
        _hairLength = 3;
    }

    public int getHairLength() {
        return _hairLength;
    }

    public void setHairLength(int length) {
        _hairLength = length;
    }

    /* bark, eat, and wagTIlle */
}
```

Accessors/Mutators

- We've filled in the `DogGroomer`'s `groom` method to modify the hair length of the `Dog` it grooms
- When a `DogGroomer` grooms a `Dog`, it calls the mutator `setHairLength` on the `Dog` and passes in 1 as an argument

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

    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Accessors/Mutators

- Can make sure `groom` method works by printing out the `Dog`'s hair length before and after we send her to the groomer
- We use accessor `getHairLength` to retrieve the value that `dogE` stores in her `_hairLength` instance variable

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

    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```
public class PetShop {
    /* This is the constructor */
    public PetShop() {
        Dog dogE = new Dog();
        DogGroomer groomer = new DogGroomer();
        System.out.println(dogE.getHairLength());
        groomer.groom(dogE);
        System.out.println(dogE.getHairLength());
    }
}

public class DogGroomer {
    public DogGroomer() {
        // this is the constructor
    }
    public void groom(Dog shaggyDog, int hairLength) {
        shaggyDog.setHairLength(hairLength);
    }
}

● What values will be printed out to the console?
● First, 3 will be printed because that’s the initial value we set for 
  _hairLength in the Dog class’s constructor
● Next, groomer sets dogE’s hair length to 1, so 1 will be printed

Example: Containment
● PetShop contains a DogGroomer
● This is a containment relationship because PetShop instantiates a 
  DogGroomer itself 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

Containment and Association
● When writing a program, need to keep in mind “big 
  picture”—how are different classes related to each other?
● Relationships between objects can be described by 
  containment and association
● Object A contains Object B when B is a component of A (A 
  creates B). Thus A knows about B and can message it. But 
  this is not symmetrical! B can’t automatically message A
● Object C and Object D are associated if C “knows about” 
  D, but D is not a component of C; this is also non-symmetric
Example: Association

- We haven't seen an association relationship yet—let's set one up!
- **Association** means that one object knows about another object that is not one of its components.

```java
class DogGroomer {
    // this is the constructor!
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- The **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
class DogGroomer {
    // this is the constructor!
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- The **PetShop** is the one who will keep track of this information about itself
- We can set up an association so that the **DogGroomer** can send her **PetShop** messages and retrieve the information she needs.

```java
class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the association
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- This is what the full association looks like
- Let's break it down line by line
- But note we're not yet making use of the association in this fragment

```java
class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the association
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
Example: Association

- We declare an instance variable named _petShop
- We want this variable to refer to the instance of PetShop that the DogGroomer belongs to

```java
public class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the instance
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- We've modified DogGroomer's constructor to take in a parameter of type PetShop
- Constructor will refer to it by the name myPetShop
- Whenever we instantiate a DogGroomer, we'll need to pass it an instance of PetShop as an argument. Which? The PetShop that created the DogGroomer

```java
public class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the instance
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- Now, we store myPetShop in the instance variable _petShop
- _petShop now points to the same PetShop instance that was passed to the constructor
- After the constructor has been executed and we can no longer reference myPetShop, we can still access the same PetShop instance by the name _petShop

```java
public class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the instance
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```

Example: Association

- Let's say we've written two accessor methods in the PetShop class that return useful information about the PetShop:
  - `getClosingTime`
  - `getNumberOfCustomers`
- If the DogGroomer ever needs these pieces of information, they can access them by calling:
  ```java
  _petShop.getClosingTime();
  _petShop.getNumberOfCustomers();
  ```

```java
public class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop; // store the instance
    }
    public void groom(Dog shaggyDog) {
        shaggyDog.setHairLength(1);
    }
}
```
public class PetShop {
    private DogGroomer _groomer;
    /* This is the constructor! */
    public PetShop() {
        _groomer = new DogGroomer(this);
        Dog dog = new Dog();
        _groomer.groom(dog);
    }
}

public class DogGroomer {
    private PetShop _petShop;
    public DogGroomer(PetShop myPetShop) {
        _petShop = myPetShop;
    }
    /* groom method slided for this example */
    public void groom(Dog dog) {
        // groom logic here
    }
}

Somewhere in memory...

The PetShop instance creates a new DogGroomer, passing itself as an argument to the DogGroomer's constructor (remember the this keyword?)

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 internal variables (a DogGroomer here).

When the DogGroomer's constructor is called, its parameter, myPetShop, points to the same PetShop that was passed in as an argument.
Another Example

The CS15Professor needs to know about 4 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 CS15App {  
  private CS15Professor _andy;  
  private HeadTA _alexa;  
  private HeadTA _marley;  
  private HeadTA _nick;  
  private HeadTA _nickie;  
  private HeadTA _allexa;  
  private HeadTA _marley;  
  private HeadTA _nick;  
  public CS15App() {  
    _alexa = new HeadTA();  
    _marley = new HeadTA();  
    _nickie = new HeadTA();  
    _nick = new HeadTA();  
    _andy = new CS15Professor(_alexa, _marley, _nickie, _nick);  
  }
}

public class CS15Professor {  
  private HeadTA _htal;  
  private HeadTA _ht2;  
  private HeadTA _ht3;  
  private HeadTA _ht4;  
  public CS15Professor(HeadTA firstTA, HeadTA secondTA, HeadTA thirdTA, HeadTA fourthTA) {  
    _htal = firstTA;  
    _ht2 = secondTA;  
    _ht3 = thirdTA;  
    _ht4 = fourthTA;  
    _htal.prepare();  
  }
  // additional methods elided*/
}

More Associations

• What if we want the Head TAs to know about CS15Professor too?  
• Need to set up another association  
• Can we just do the same thing?
More Associations

- This doesn't work: when we instantiate _alexa, _marley, _nickie and _nick, we try to pass them _andy
- But _andy hasn't been instantiated yet!
- What can we try instead?

```java
public class CS15App {
    private CS15Professor _andy;
    private HeadTA _alexa;
    private HeadTA _marley;
    private HeadTA _nickie;
    private HeadTA _nick;
    public CS15App() {
        _alexa = new HeadTA(_andy);
        _marley = new HeadTA(_andy);
        _nickie = new HeadTA(_andy);
        _nick = new HeadTA(_andy);
        _andy = new CS15Professor(_alexa, _marley, _nickie, _nick);
    }
}
```

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.
Packages

- Several core packages already exist as part of the Java language
- They contain classes that will be very useful to you in future projects! For example:
  - package javafx.scene.paint contains a class called Color, which you will use to make your programs pretty!
  - package javafx.scene.shape contains classes like Ellipse and Rectangle that you will use to build shapes
  - package java.lang contains a class called Math, which contains methods for numeric operations like square root and absolute value

Working Within a Package

- Let's say that Dog, DogGroomer and PetShop are all in a package called PetShopProject
- At the top of each class's file, we need to specify that it is contained in the PetShopProject package

```java
package PetShopProject;
public class Dog{
    public Dog() {
        // this is the constructor!
    }
    /* other methods elided */
}
```
Using Code from Another Package

package PetShopProject;
public class DogCollar {
    private javafx.scene.paint.Color _color;
    public DogCollar() {
        _color = new javafx.scene.paint.Color(255, 0, 0);
    }
    public void setColor(javaFX.scene.paint.Color newColor) {
        _color = newColor;
    }
    public javafx.scene.paint.Color getColor() {
        return _color;
    }
}

 constructor of DogCollar
instantiates a javafx.scene.paint.Color and stores it in instance variable _color

public class DogCollar {
    private javafx.scene.paint.Color _color;
    public DogCollar() {
        _color = new javafx.scene.paint.Color(255, 0, 0);
    }
    public void setColor(javaFX.scene.paint.Color newColor) {
        _color = newColor;
    }
    public javafx.scene.paint.Color getColor() {
        return _color;
    }
}

Using Code from Another Package

Here's an example of using the Color class from the package javafx.scene.paint.

package PetShopProject;
public class DogCollar {
    private javafx.scene.paint.Color _color;
    public DogCollar() {
        _color = new javafx.scene.paint.Color(255, 0, 0);
    }
    public void setColor(javaFX.scene.paint.Color newColor) {
        _color = newColor;
    }
    public javafx.scene.paint.Color getColor() {
        return _color;
    }
}

DogCollar class keeps track of its color as an instance variable named _color

public class DogCollar {
    private javafx.scene.paint.Color _color;
    public DogCollar() {
        _color = new javafx.scene.paint.Color(255, 0, 0);
    }
    public void setColor(javaFX.scene.paint.Color newColor) {
        _color = newColor;
    }
    public javafx.scene.paint.Color getColor() {
        return _color;
    }
}

We've created accessor and mutator methods for _color

setColor takes in a javafx.scene.paint.Color, and assigns the instance of Color it receives as a parameter to instance variable _color

getcolor returns current instance of Color that _color refers to

We've created accessor and mutator methods for _color

setColor takes in a javafx.scene.paint.Color, and assigns the instance of Color it receives as a parameter to instance variable _color

getcolor returns current instance of Color that _color refers to
Import Statements

- It would be pretty annoying to have to write out the full package name every time we use `Color` class.
- If we use an import statement at the top of the file to import the class `javafx.scene.paint.Color`, we can simply refer to it as `Color` from then on.
- This makes code more concise and readable.

```java
package PetShopProject;
public class DogCollar {
    private javafx.scene.paint.Color _color;
    public DogCollar() {
        _color = new javafx.scene.paint.Color(255, 0, 0);
    }
    public void setColor(javafx.scene.paint.Color newColor) {
        _color = newColor;
    }
    public javafx.scene.paint.Color getColor() {
        return _color;
    }
}
```

With Import Statement

```java
package PetShopProject;
import javafx.scene.paint.Color;
public class DogCollar {
    private Color _color;
    public DogCollar() {
        _color = new Color(255, 0, 0);
    }
    public void setColor(Color newColor) {
        _color = newColor;
    }
    public Color getColor() {
        return _color;
    }
}
```

Summary

- Let's review the three types of properties:
  - Attributes: descriptors of an object (e.g., `Color` of a `DogCollar`)
  - Components: "parts" of an object (e.g., `CS15Professor` is a component of `CS15App`)
  - Associations: things that are not part of an object, but the object needs to know about (e.g., `CS15Professor` needs to know about `CS15HeadTAs`)
- Attributes and components are created within a class, while associations are between peer objects or from a component to its container.
- These are distinctions we make conceptually – Java just knows about instance variables!

That's it!

Important concepts:
- Using local variables, which exist within 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
- Containment: when one object is a component of another so the container can therefore send the component it created messages
- Association: when one object knows about another object that is not one of its components—has to be set up explicitly
- Using import statements to access code from different packages
Announcements (1/2)

• AndyBot is due **tonight at 11:59PM** - no late handin
• We have sent out grades for HW1 - contact HTAs if you haven’t heard back
• Lab 1 is released today
  - If you haven’t been checked off for lab0, you can do so at TA hours until Thursday

Announcements (1/2)

• You are responsible for the information in all emails we send you and public piazza answers
• Remember to submit Piazza questions *privately* - or else you risk violating the collab policy.
  - Also be sure to check that your question hasn’t already been answered!
• Lastly, if you registered within the last few days, come see the HTAs *ASAP*