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

Review: Methods

- **Call methods**: send messages to an object

```
andyBot.turnRight();
```

- **Define methods**: give a class specific capabilities

```
public void turnLeft() {
    // code to turn Robot left goes here
}
```


Review: Constructors and Instances

- Declare a **constructor** (a method called whenever an object is "born")

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

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

```
new Calculator();
```


Review: Parameters and Arguments

- Write methods that take in **parameters** (input) and have **return** values (output), e.g., this `Calculator`'s method

```
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)

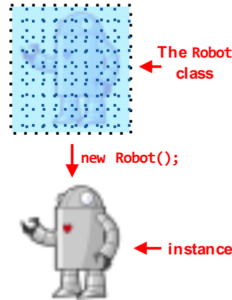
```
myCalculator.add(5, 8);
```


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 building 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 `Dog`s know how to bark, eat, and wag their tails

```
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:

```
new Dog();
```

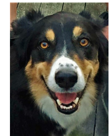
```
public class Dog {

  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
  }
}
```


Variables

- Once we create a `Dog`, 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 object by storing it in a **variable**

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

/* named after Django Reinhardt – see https://www.youtube.com/watch?v=plpSfvdCH0Q */



- A **variable** stores information
- In this case, `django` is the variable, and it stores a newly created instance of `Dog` somewhere in memory

Syntax: Variable Declaration and Assignment

- To declare and assign a variable, thereby initializing it, in a single statement is: `Dog django = new Dog();`

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

- Note: type of `value` must match declared `type` on left
- Note that we can reassign as many times as we like (example soon)

Variables

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

- The "=" operator **assigns** the instance of `Dog` that we created to the variable `django`. We say "`django` gets a new `Dog`"
- Now we can call methods on our `Dog` using its new name (`django`), e.g., `django.bark();`

django

Assignment vs. Equality

In Java:

```
price= price + 10;
```

- Means "add 10 to the current value of price and assign that to price"

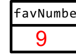
In Algebra:

- price = price + 10 is a logical contradiction

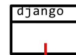
Variables Store Information: Values vs. References

- A variable stores information as either:
  - a *value of a primitive (aka base) type* (like `int` or `float`)
  - or 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 objects vary in size, hence Java simplifies its memory management by having a fixed size reference to an instance elsewhere in memory
  - "one level of indirectness"

`int favoriteNumber = 9;`



`Dog django = new Dog();`



(somewhere else in memory)

Clicker Question

Given this code, fill in the blanks:

```
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 {

  /*constructor of trivial PetShop! */
  public PetShop() {
    this.testDjango();
  }

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

  }
    …
}
```

- Let's call the `testDjango()` method within the constructor of the `PetShop` class
- Whenever someone instantiates a `PetShop`, it in turn calls `testDjango()`, which in turn instantiates a `Dog`
- Then it tells the `Dog` to bark, eat, and wag its tail

Example: Instantiation (2/2)

```
public class MathStudent {

  /* constructor elided */

  public void performCalculation() {
    Calculator myCalc = new Calculator();
    int answer = myCalc.add(2, 6);
    System.out.println(answer);
  }
       …
}
```

- *Another example:* can instantiate a a `MathStudent` and then call that instance to perform a simple, fixed, calculation
- 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 (1/4)

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

```
public class DogGroomer {

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

  public void groom(Dog shaggyDog) {
    // code that grooms shaggyDog
  }
}
```

Objects as Parameters (2/4)

- DogGroomer's groom method takes in a single parameter– a Dog
- Always specify **type**, then **name** of parameter
- Here, Dog is type and "shaggyDog" is name (aka dummy/symbolic parameter) we've chosen – whatever reference to a dog is passed in is called shaggyDog in this method
- Note that in algebra, we only have numeric types, so no need to "declare" type explicitly

```
public class DogGroomer {

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


Andries van Dam © 2016 09/20/16

Objects as Parameters (3/4)

- How to call the groom method?
- Do this in the PetShop helper method testGroomer()
- PetShop's call to testGroomer() instantiates a Dog and a DogGroomer, then tells the DogGroomer to groom the Dog

```
public class PetShop {

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

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

}
```


Andries van Dam © 2016 09/20/16

Objects as Parameters (4/4)

- 0. Elsewhere in the program, some method instantiates a PetShop (thereby calling PetShop's constructor). Then:
  1. The PetShop in turn calles 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 groom method is called on groomer, passing in django as an argument; the groomer will think of it as shaggyDog, a synonym

```
public class App {
  public App() {
  0.Petshop petSmart = new Petshop();
  }
}

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

  public void testGroomer() {
  1.Dog django = new Dog();
  2.DogGroomer groomer = new DogGroomer();
  3.groomer.groom(django);

  }
}
```


Andries van Dam © 2016 09/20/16

What is Memory?

- Memory (system memory, not disk or other peripheral devices) is the hardware in which computers store information, both temporary and permanent
- Think of memory as a list of slots; each slot holds information (e.g., a local 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

```
//Elsewhere in the program
Petshop petSmart = new Petshop();

public class PetShop {

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

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

}
```


Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (1/6)

```
public class PetShop {

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

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

```
public class DogGroomer {

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

  public void groom(Dog shaggyDog) {
    // code that grooms shaggyDog goes here!
  }
}
```

Somewhere in memory ...

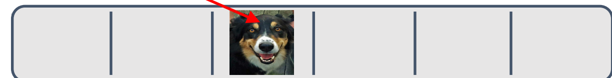
Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (2/6)

```
public class PetShop {

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

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

```
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 django to refer to this particular Dog, at this particular location in memory.

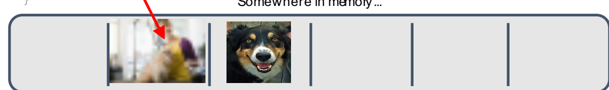
Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (3/6)

```
public class PetShop {

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

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

```
public class DogGroomer {

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

  public void groom(Dog shaggyDog) {
    // code that grooms shaggyDog goes here!
  }
}
```

Somewhere in memory ...



The same goes for the DogGroomer—we store a particular DogGroomer somewhere in memory. Our PetShop knows this DogGroomer by the name groomer.

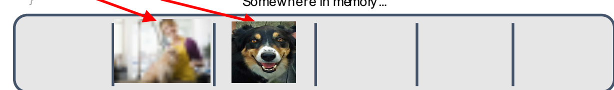
Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (4/6)

```
public class PetShop {

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

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

```
public class DogGroomer {

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

  public void groom(Dog shaggyDog) {
    // code that grooms shaggyDog goes here!
  }
}
```

Somewhere 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 in a parameter of type Dog). We tell her to groom django.

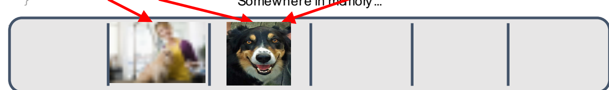
Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (5/6)

```
public class PetShop {

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

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

```
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 in django 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.

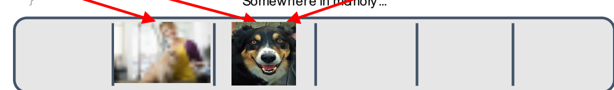
Andries van Dam © 2016 09/20/16

Objects as Parameters: Under the Hood (6/6)

```
public class PetShop {

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

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

```
public class DogGroomer {

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

  public void groom(Dog shaggyDog) {
    // code that grooms shaggyDog goes here!
  }
}
```

Somewhere in memory ...



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.

Andries van Dam © 2016 09/20/16

Variable Reassignment (1/2)

- 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 re-use the variable django to first point to one Dog, then another!

```
public class PetShop {

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

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

}
```


Andries van Dam © 2016 09/20/16

Variable Reassignment (2/2)

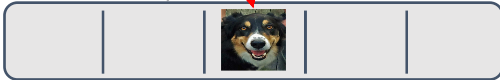
- First, instantiate another Dog, and reassign variable django to point to it
- Now django no longer refers to the first Dog instance we created, which has already been groomed
- We then tell groomer to groom the newer Dog

```
public class PetShop {

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

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


Andries van Dam © 2016 09/20/16

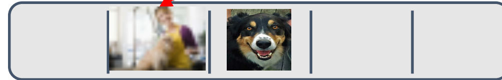
Variable Reassignment: Under the Hood (1/5)

```
public class PetShop {

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

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



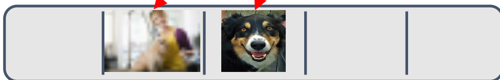
Variable Reassignment: Under the Hood (2/5)

```
public class PetShop {

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

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



Variable Reassignment: Under the Hood (3/5)

```
public class PetShop {

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

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



Variable Reassignment: Under the Hood (4/5)

```
public class PetShop {

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

  public void testGroomer() {
    Dog django = new Dog();
    DogGroomer groomer = new DogGroomer();
    groomer.groom(django);
    django = new Dog(); //old ref garbage collected
    groomer.groom(django);
  }
```

Variable Reassignment: Under the Hood (5/5)

```
public class PetShop {

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

  public void testGroomer() {
    Dog django = new Dog();
    DogGroomer groomer = new DogGroomer();
    groomer.groom(django);
    django = new Dog();//old ref garbage collected
    groomer.groom(django);
  }
```

Clicker Question

What is the correct value of `(a+b)` after the following code is executed?

```
int a = 3;
int b = 2;
a = b + 2;
b = a + 1;
```

A. 5
B. 9
C. 7
D. 6

Local Variables (1/2)

- All variables we've seen so far have been **local variables**: variables declared *within 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
  - the same is true of method parameters

```
public class PetShop {

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

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

local variables

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 :(
  - "Garbage Collection" – stay tuned


```
public class PetShop {

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

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

local variables

Accessing Local Variables

- If you try to access a local variable outside of it's method, you'll receive a "cannot find symbol" compilation error.

In Terminal:

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

```
public class PetShop {

  private DogGroomer _groomer;

  /* This is the constructor! */
  public PetShop() {
    _groomer = new DogGroomer();
    Dog django = new Dog();
  }

  public void exerciseDjango() {
    django.playCatch();.
  }

}
```


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, not within a single method, and are accessible from anywhere within the class – its 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 object (instance of some other class)

Modeling Properties with Instance Variables (1/2)

- 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 (2/2)

- All instances of a class have the same properties, but the *values* of these properties will differ
- All CS15Students 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 should be instance variables
  - **associations**: things that are not part of an object, but that the object needs to know about. For example, the instructor needs to know about his/her TAs (more on this soon)
- All methods in a class can access all of its properties, to use them and/or to change them

Instance Variables (1/4)

- We've modified PetShop example to make our DogGroomer an **instance variable**
- Split up declaration and assignment of instance variable:
  - **declare** instance variable at the top of the class, to notify Java
  - **initialize** the instance variable by **assigning** a value to it in the constructor
  - purpose of constructor is to initialize all instance variables so the instance has a valid initial "state" at its "birth"
  - state is the set of all values for all properties—local variables don't hold properties - they are "temporaries"

```
public class PetShop {        declaration

  private DogGroomer _groomer;

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

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


Instance Variables (2/4)

- 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

```
                              access modifier
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.groom(django);
  }
}
```


Instance Variables (3/4)

- 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**
- Note that local variables don't have access modifier– they always have the same scope (their own method)

```
                              access modifier
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.groom(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, and you publish only those properties you want others to have access to (stay tuned...)

```
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.groom(django);
  }
}
```

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() {
        //oops!
        this.testGroomer();
    }

    public void testGroomer() {
        Dog django = new Dog(); //local var
        _groomer.groom(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"!
- 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;

    public PetShop() {
        //oops!
        this.testGroomer();
    }

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

Instance Variables (1/2)

- Let's add an instance variable to the `Dog` class
- `_hairLength` is an `int` that will keep track of the 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 wagTail elided */
}
```


Instance Variables (2/2)

- `_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; /* all dogs have same hairlenghth initially */
    }

    /* bark, eat, and wagTail elided */
}
```


Accessors/Mutators

- The class may make the value of an instance variable publicly available via an **accessor method** that **returns** the value when called
- `getHairLength` is an accessor method for `_hairLength`
- Can call `getHairLength` on an instance of Dog to **return** its current `_hairLength` value
- Remember: the return type you specify and the value you return must match!

```
public class Dog {

    private int _hairLength;

    public Dog() {
        _hairLength = 3;
    }

    public int getHairLength() {
        return _hairLength;
    }

    /* bark, eat, and wagTail elided */
}
```


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`

```
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 wagTail elided */
}
```

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

```
public class DogGroomer {

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

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


Example: Accessors(1/2)

- Can make sure groom method works by printing out the Dog's hair length before and after we send it to the groomer

```
public class PetShop {
  private DogGroomer _groomer;

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

  public void testGroomer() {
    Dog django = new Dog();
    System.out.println(django.getHairLength());
    _groomer.groom(django);
    System.out.println(django.getHairLength());
  }
}
```

```
public class DogGroomer {

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

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

- We use **accessor** getHairLength to retrieve the value that django stores in its _hairLength instance variable

Example: Accessors(2/2)

- What values will be printed 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.getHairLength());
    _groomer.groom(django);
    System.out.println(django.getHairLength());
  }
}
```

```
public class DogGroomer {

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

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

- First, 3 will be printed because that's the initial value we set for _hairLength in the Dog class's constructor
- Next, groomer sets django's hair length to 1, so 1 will be printed

Example: Mutators

- What if we don't always want to cut the dog's hair to a length of 1?
- When we tell groomer to groom, let's also tell groomer how short to cut the hair

```
public class PetShop {

  // Constructor elided

  public void testGroomer() {
    Dog django = new Dog();
    _groomer.groom(django, 2);
  }
}
```

```
public class DogGroomer {
  /* Constructor and other code elided */

  public void groom(Dog shaggyDog, int hairLength) {
    shaggyDog.setHairLength(hairLength);
  }
}
```

The groomer will cut the dog's hair to a length of 2!

- groom will take in another parameter, and set dog's hair length to value of hairLength
- Now pass two parameters when we call the groom method so that the _groomer knows how long hairLength should be

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** or **association**
- Object A **contains** Object B when B is a component of A (A creates B). Thus A knows about B and can call methods on it. But this is not symmetrical! B can't automatically call methods on 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: Containment

- PetShop **contains** a DogGroomer
- Containment relationship because PetShop itself instantiates a DogGroomer 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

```
public class PetShop {

  private DogGroomer _groomer;

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

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

Example: Association (1/8)

- 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

```
public class DogGroomer {

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

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


Example: 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…

```
public class DogGroomer {

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

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


Example: Association (3/8)

- The PetShop keeps track of such information in its properties
- Can set up an **association** so that DogGroomer can send her PetShop messages to retrieve information she needs

```
public class DogGroomer {

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

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


Example: Association (4/8)

- 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

```
public class DogGroomer {

  private PetShop _petShop;

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

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


Example: Association (5/8)

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

```
public class DogGroomer {

  private PetShop _petShop;

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

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


Example: Association (6/8)

- 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 instance that created the DogGroomer, hence use this

```
public class DogGroomer {
   private PetShop _petShop;

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

```
public class PetShop {
   private DogGroomer _groomer;

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

   //testGroomer() elided
}
```

Example: Association (7/8)

- Now store `myPetShop` in instance variable `_petShop`
- `_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 asso
  }

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


Example: 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)`

```
public class DogGroomer {

  private PetShop _petShop;
  private Time _closingTime;

  public DogGroomer(PetShop myPetShop) {
    _petShop = myPetShop; // store assoc.
    _closingTime = myPetShop.getClosingTime();
    _petShop.setNumCustomers(10);
  }
}
```


Association: Under the Hood (1/5)

```
public class PetShop {
  private DogGroomer _groomer;

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

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

```
public class DogGroomer {
  private PetShop _petShop;

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

  /* groom and other methods elided for this
  example */
}
```

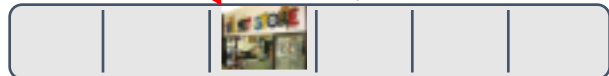
Somewhere in memory ...

Association: Under the Hood (2/5)

```
public class PetShop {
  private DogGroomer _groomer;

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

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

```
public class DogGroomer {
  private PetShop _petShop;

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

  /* groom and other methods elided for this
  example */
}
```

Somewhere in memory ...



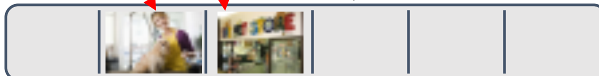
Somewhere else in our code, someone calls new PetShop(). An instance of PetShop is created somewhere in memory and PetShop's constructor initializes all its instance variables (just a DogGroomer here)

Association: Under the Hood (3/5)

```
public class PetShop {
  private DogGroomer _groomer;

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

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

```
public class DogGroomer {
  private PetShop _petShop;

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

  /* groom and other methods elided for this
  example */
}
```

Somewhere in memory ...



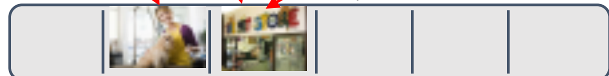
The PetShop instantiates a new DogGroomer, passing itself in as an argument to the DogGroomer's constructor (remember the this keyword?)

Association: Under the Hood (4/5)

```
public class PetShop {
  private DogGroomer _groomer;

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

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

```
public class DogGroomer {
  private PetShop _petShop;

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

  /* groom and other methods elided for this
  example */
}
```

Somewhere in memory ...



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

Association: Under the Hood (5/5)

```
public class PetShop {
    private DogGroomer _groomer;

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

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

```
public class DogGroomer {
    private PetShop _petShop;

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

    /* groom and other methods elided for this
    example */
}
```

Somewhere in memory...

The DogGroomer sets its _petShop instance variable to point to the same PetShop it received as an argument. Now it "knows about" the petShop that instantiated it! And therefore so do all its methods...

Andries van Dam © 2016 09/20/16

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 – they are peer objects
- 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 */
}
```


Andries van Dam © 2016 09/20/16

Another Example: Association (2/6)

- 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 CS15Professor {

    // declare instance variables here
    // and here…
    // and here…
    // and here!

    public CS15Professor(/* parameters */) {

        // initialize instance variables!
        // …
        // …
        // …
    }

    /* additional methods elided */
}
```


Andries van Dam © 2016 09/20/16

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:

  _hta2.setUpLecture();

```
public class CS15Professor {

    private HeadTA _hta1;
    private HeadTA _hta2;
    private HeadTA _hta3;
    private HeadTA _hta4;

    public CS15Professor(HeadTA firstTA,
          HeadTA secondTA, HeadTA thirdTA
          HeadTA fourthTA) {

        _hta1 = firstTA;
        _hta2 = secondTA;
        _hta3 = thirdTA;
        _hta4 = fourthTA;
    }

    /* additional methods elided */
}
```


Andries van Dam © 2016 09/20/16

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.

```
public class CS15App {

    // declare CS15Professor instance var.
    // declare four HeadTA instance vars.
    // …
    // …
    // …

    public CS15App() {
        // instantiate the four HeadTAs
        // …
        // …
        // instantiate the professor!
    }
}
```


Andries van Dam © 2016 09/20/16

Another Example: Association (5/6)

- We declare _andy, _dan, _divya, _emily and _sophia as instance variables
- In the constructor, we instantiate them
- Since the constructor of CS15Professor takes in 4 HeadTAs, we pass in _dan, _divya, _emily and _sophia

```
public class CS15App {

    private CS15Professor _andy;
    private HeadTA _dan;
    private HeadTA _divya;
    private HeadTA _emily;
    private HeadTA _sophia;

    public CS15App() {
        _dan = new HeadTA();
        _divya = new HeadTA();
        _emily = new HeadTA();
        _sophia = new HeadTA();
        _andy = new CS15Professor(_dan,
                _divya, _emily, _sophia);
    }
}
```


Andries van Dam © 2016 09/20/16

Another Example: Association (6/6)

```
public class CS15Professor {

  private HeadTA _hta1;
  private HeadTA _hta2;
  private HeadTA _hta3;
  private HeadTA _hta4;

  public CS15Professor(HeadTA firstTA,
        HeadTA secondTA, HeadTA thirdTA
        HeadTA fourthTA){

    _hta1 = firstTA;
    _hta2 = secondTA;
    _hta3 = thirdTA;
    _hta4 = fourthTA;
    _hta2.prepLecture();
  }
  /* additional methods elided */
}
```

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _dan;
  private HeadTA _divya;
  private HeadTA _emily;
  private HeadTA _sophia;

  public CS15App() {
    _dan = new HeadTA();
    _divya = new HeadTA();
    _emily = new HeadTA();
    _sophia = new HeadTA();
    _andy = new CS15Professor(_dan,
          _divya, _emily, _sophia);
  }
}
```

79/94

More Associations (1/5)

- 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?

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _dan;
  private HeadTA _divya;
  private HeadTA _emily;
  private HeadTA _sophia;

  public CS15App() {
    _dan = new HeadTA();
    _divya = new HeadTA();
    _emily = new HeadTA();
    _sophia = new HeadTA();
    _andy = new CS15Professor(_dan,
          _divya, _emily, _sophia);
  }
}
```

80/94

More Associations (2/5)

- This doesn't work: when we instantiate _dan, _divya, _emily and _sophia, we would like to pass them an argument, _andy
- But _andy hasn't been instantiated yet! And can't initialize _andy first because the headTAs haven't been created yet…
- What can we try instead?

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _dan;
  private HeadTA _divya;
  private HeadTA _emily;
  private HeadTA _sophia;

  public CS15App() {
    _dan = new HeadTA();
    _divya = new HeadTA();
    _emily = new HeadTA();
    _sophia = new HeadTA();
    _andy = new CS15Professor(_dan,
          _divya, _emily, _sophia);
  }
}
```

81/94

More Associations (3/5)

- Need a way to pass _andy to _dan, _divya, _emily and _sophia **after** we instantiate _andy
- Use a new method, setProf, and pass each Head TA _andy

```
public class CS15App {

  private CS15Professor _andy;
  private HeadTA _dan;
  private HeadTA _divya;
  private HeadTA _emily;
  private HeadTA _sophia;

  public CS15App() {
    _dan = new HeadTA();
    _divya = new HeadTA();
    _emily = new HeadTA();
    _sophia = new HeadTA();
    _andy = new CS15Professor(_dan,
          _divya, _emily, _sophia);

    _dan.setProf(_andy);
    _divya.setProf(_andy);
    _emily.setProf(_andy);
    _sophia.setProf(_andy);
  }
}
```

82/94

More Associations (4/5)

```
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 _dan;
  private HeadTA _divya;
  private HeadTA _emily;
  private HeadTA _sophia;

  public CS15App() {
    _dan = new HeadTA();
    _divya = new HeadTA();
    _emily = new HeadTA();
    _sophia = new HeadTA();
    _andy = new CS15Professor(_dan,
          _divya, _emily, _sophia);

    _dan.setProf(_andy);
    _divya.setProf(_andy);
    _emily.setProf(_andy);
    _sophia.setProf(_andy);
  }
}
```

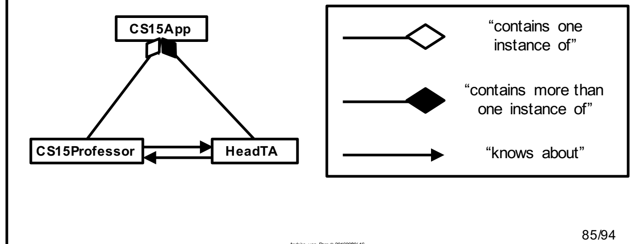
83/94

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

84/94

Visualizing Containment and Association



Clicker Question

Is this a valid way to associate `Teacher` and `School`?

```
public class School{
  private Teacher _teacher;
  public School() {
  _teacher = new Teacher(this);
  this.assignTeacher();
  }
```

```
public class Teacher{
  private School _school;

  public Teacher(School school) {
  _school = school;
  }
```

A. Yes B. No

Summary

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

Announcements

- AndyBot is due **tonight** at 11:59pm- no late handin
  - Please remember to run `cs015_handin AndyBot`
    - Just having the files in the directory is not enough
- Lab0 is due by the end of your lab this week, Lab1 is out now
- Please only post private questions on Piazza
  - TAs will make the question public if they think it will benefit the class
- FastX issues? See the note on Piazza about X Forwarding and SSH