Lecture 4

Working with Objects: Part 1
Outline

- Storing values in variables
- Instances as parameters
- Variable reassignment
- Delegation pattern and containment
- Local variables vs. instance variables
Review: Methods

- **Call methods**: give 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: Parameters and Arguments

- **Define** methods that take in generic **parameters** (input) and have **return** values (output); e.g., this **Calculator**’s method:
  ```java
  public int add(int x, int y) {
      return x + y; // x, y are dummy (symbolic) variables
  }
  ```
- **Call** such methods on instances of a class by providing specific **arguments** (actual values for symbolic parameters)
  ```java
  myCalculator.add(5, 8);
  ```
- **Remember** the one-to-one correspondence rule: list of arguments must match list of parameters in number, order, and types
  - thus, Java can substitute each argument for its corresponding parameters
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
  - 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 using the `new` keyword:
  ```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
      }
  }
  ```
  ```java
  new Dog();
  ```
Outline

- Storing values in variables
- Instances as parameters
- Variable reassignment
- Delegation pattern and containment
- Local variables vs. instance variables
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
  
  ```javascript
  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

- We can both declare and assign (i.e., initialize) a variable in a single statement, like: `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"

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

- We can reassign a variable 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”

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();
```
TopHat 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)

- 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` for what these methods do)
Another 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);
    }

    /* add() method elided */
    ...
}

• Another example: can instantiate a MathStudent and then call that instance to perform a simple, fixed, calculation, called performCalculation()

• First, instantiate a 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!
Outline

- Storing values in variables
- Instances as parameters
- Variable reassignment
- Delegation pattern and containment
- Local variables vs. instance variables
Instances as Parameters (1/3)

- Methods can take in not just numbers but also instances as parameters.
- The `PetShop` 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 PetShop {
    public PetShop() {
        // 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 PetShop's trimFur method?
- Do this in the App method testGrooming()
- Call to testGrooming() instantiates a PetShop and a Dog, then calls the PetShop to trimFur of the Dog
- First two lines could be in either order, since both are instantiated adjacently

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }
    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
    }
}
```
0. In App’s `main` method, call to `testGrooming()` helper method.

Then:

1. A `PetShop` is instantiated (thereby calling `PetShop`’s constructor) and a reference to it is stored in the variable `andysPetShop`

2. Next, a `Dog` is instantiated (thereby calling `Dog`’s constructor) and a reference to it is stored in the variable `django`

3. The `trimFur` method is called on `andysPetShop`, passing in `django` as an argument

4. `andysPetShop` trims `django`’s fur; `trimFur` in `andysPetShop` will think of `django` as `shaggyDog`, a synonym

```java
public class App {
    public static void main(String[] args) {
        0. this.testGrooming();
    }

    public void testGrooming() {
        1. PetShop andysPetShop = new PetShop();
        2. Dog django = new Dog();
        3. andysPetShop.trimFur(django);
            //exit method, django and groomer disappear
    }
}
```

```java
public class PetShop {
    /* constructor elided */

    public void trimFur(Dog shaggyDog) {
        4. // code that trims the fur of shaggyDog
    }
}
```
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 `PetShop` instance.
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 `PetShop`, it's stored somewhere in memory. Our `App` will use the name `andysPetShop` to refer to this particular `PetShop`, at this particular location in memory.
Same goes for the Dog—we store a particular Dog somewhere in memory. Our App knows this Dog by the name django.
We call the `trimFur` method on our `PetShop`, `andysPetShop`. We need to tell it which `Dog` to `trimFur` (since the `trimFur` method takes in a parameter of type `Dog`). We tell it 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`. Usually not adjacent in memory!
Instances as Parameters: Under the Hood (6/6)

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
    }
}
```

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

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

Somewhere in memory...

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`. Usually not adjacent in memory!
Outline

● Storing values in variables
● Instances as parameters
● Variable reassignment
● Delegation pattern and containment
● Local variables vs. instance variables
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 PetShop to **trimFur** two different Dogs?

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

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.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 new `Dog`. It will also be known as `shaggyDog` inside the `trimFur` method.

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog(); // reassign django
        andysPetShop.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; // reassigns django to refer to the same Dog as scooby
```

- Now `django` and `scooby` refer to the same `Dog`, specifically the one that was originally referenced by `scooby`
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog();
        andysPetShop.trimFur(django);
    }
}
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog();
        andysPetShop.trimFur(django);
    }
}
Variable Reassignment: Under the Hood (3/5)

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog();
        andysPetShop.trimFur(django);
    }
}
```
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog(); //old ref garbage collected - stay tuned!
        andysPetShop.trimFur(django);
    }
}
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        PetShop andysPetShop = new PetShop();
        Dog django = new Dog();
        andysPetShop.trimFur(django);
        django = new Dog(); // old ref garbage collected - stay tuned!
        andysPetShop.trimFur(django);
    }
}
Outline

● Storing values in variables
● Instances as parameters
● Variable reassignment
● Delegation pattern and containment
● Local variables vs. instance variables
Adding PetShop Capabilities

• The PetShop only has the capability (method) to trimFur

• What if we want the PetShop to expand with more functionality?

• PetShop class would be long!

  • trimFur
  • shampooFur
  • dryFur
  • teachSit
  • teachBark
  • teachFetch
  • sellDogToy
  • and more…
Delegation Pattern (1/3)

• Just like a real-life pet shop would hire employees to delegate work, we should create new classes to delegate code

• Pass responsibility to something / someone else to manage parts of a task

• **PetShop** doesn’t need to care *how* the dog gets trimmed, if it gets done properly
Delegation Pattern (2/3)

• Delegation results in a **chain of abstraction**, where each level deals with more specifics to complete an action

Please groom my dog!

Wash this dog with shampoo, then trim its hair and dry!

Fill the bath with warm water until it’s two-thirds full…

DogOwner → PetShop → DogGroomer

Bath → HairDryer → Clippers
Delegation Pattern (3/3)

- We delegate responsibilities to **DogGroomer**!

- **trimFur** becomes a capability of **DogGroomer** instead of **PetShop**

- **teachSit** and **teachBark** can be delegated to **DogTrainer**

```java
public class DogGroomer {
  /* constructor elided */
  public void trimFur(Dog shaggyDog) {
    //code that trims the fur of shaggyDog
  }

  public void shampooFur(Dog dirtyDog) {
    //code that shampoos the fur of dirtyDog
  }

  public void dryFur(Dog wetDog) {
    //code that dries the fur of wetDog
  }
}
```
Aside: Design Patterns and Principles

- **Delegation** is the first *design pattern* we’re learning
- We’ll learn many throughout the course – these are crucial to OOP
- OOP is about much more than *functionality* of programs
  - *PetShop* could operate fine without *DogGroomer* or *DogTrainer*; delegating is our design choice to make code easier to read
- Later, assignment grades will be based as much on your design choices as functionality
- In future projects, YOU will have to decide how to delegate your program to different classes!
  - (not quite yet though)
Consequence of Delegation

• With delegation, we’ll use multiple classes to accomplish one task
  - PetShop, DogGroomer, Bath, HairDyer, and Clippers all involved with dog grooming

• Must ask ourselves - How are different classes related to each other so their instances can communicate to collaborate?

• Two key concepts to establish these relationships are containment and association
Containment

- Often a class A will need as a component an instance of class B, so A will create the instance of B by using the `new` keyword.

- Any time class A creates a new instance of class B, we say A **contains** that instance of class B.

- A knows about B and can call B’s methods on that instance.

- Note this is **not symmetrical**: B can’t call methods on A!
  
  - thus, a **Car** can call methods on a contained instance of **Engine**, but the **Engine** instance can’t call methods on the **Car** instance – it doesn’t know about the **Car** instance that it is contained in.
Visualizing Containment

- Notation comes from UML (Unified Modeling Language) standards used to model software systems
Example: Containment

- Now that we’ve delegated responsibilities to the `DogGroomer`, the `PetShop` can **contain** an instance of `DogGroomer`.

- In this method, `PetShop` can call `DogGroomer`’s methods on `groomer`.

- It may seem unnatural to have a `PetShop` contain a `DogGroomer`, but it works in the kind of modeling that OOP makes possible.

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

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

(Notice the methods being called on `groomer` are defined in `DogGroomer`).
Delegating to Top-Level Class (1/2)

**App** class should never have more than a few lines of code

```java
public class App {
    public static void main(String[] args) {
        this.testGrooming();
    }

    public void testGrooming() {
        DogGroomer groomer = new DogGroomer();
        Dog django = new Dog();
        groomer.shampooFur(django);
        groomer.trimFur(django);
        groomer.dryFur(django);
    }
}
```
Delegating to Top-Level Class (2/2)

• **Top-level class** is class that contains high-level logic of program

• **App delegates** to **top-level class** (here, **PetShop**) to simplify **App** as much as possible

• Same **functionality** of the program, with a different **code design**
  - easier to visually follow program’s high-level control flow

• As **CS15** programs increase in complexity, purpose of separating top-level class from **App** will become clearer

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

public class PetShop {
    public PetShop() {
        this.testGrooming();
    }
    public void testGrooming() {
        DogGroomer groomer = new DogGroomer();
        Dog django = new Dog();
        groomer.shampooFur(django);
        groomer.trimFur(django);
        groomer.dryFur(django);
    }
}
TopHat Question

Which of the following is NOT true?

A. App should delegate to the top-level class
B. The top-level class should never have more than a few lines of code
C. App should contain the top-level class
D. The relationship between App and the top-level class can be visualized as:

```
      App
        ▲
        ▽
  Top-Level Class
```
Outline

- Storing values in variables
- Instances as parameters
- Variable reassignment
- Delegation pattern and containment
- Local variables vs. instance variables
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.

```
public class PetShop {
  public PetShop() {
    this.testGrooming();
  }

  public void testGrooming() {
    Dog django = new Dog();
    DogGroomer groomer = new DogGroomer();
    groomer.shampooFur(django);
    groomer.trimFur(django);
    groomer.dryFur(django);
  }
}
```
Local Variables (2/2)

- We created `groomer` and `django` in our `PetShop`'s `testGrooming` method, but as far as the rest of the class is concerned, they don’t exist and cannot be used.

- Once the method is completely executed, they’re gone:(
  - this is known as “Garbage Collection”

```java
public class PetShop {

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

    public void testGrooming() {
        Dog django = new Dog();
        DogGroomer groomer = new DogGroomer();
        groomer.shampooFur(django);
        groomer.trimFur(django);
        groomer.dryFur(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:
  - at the end of method execution, local variables created within that method go out of scope
  - 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 {
    public PetShop() {
        DogGroomer groomer = new DogGroomer();
        this.cleanShop();
    }

    public void cleanShop() {
        //assume we’ve added a sweep method
        //to DogGroomer
        groomer.sweep();
    }
}
```

In Terminal after `javac *.java`:
```
PetShop.java:13: error: cannot find symbol
groomer.sweep();
^ symbol: variable groomer
location: class PetShop
```
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.
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 therefore are accessible from anywhere within the class, unlike local variables – 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 **values** of those properties can be different and can differentiate one 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 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
  - All **CS15Students** have a “height”, but the value stored in instance variable would differ from instance to instance
Instance Variables (1/4)

- We’ve modified PetShop example to make our DogGroomer an instance variable for the benefit of multiple methods

- Split up declaration and assignment of instance variable:
  - declare instance variable at the top of the class above the constructor, to notify Java compiler
  - initialize the instance variable by assigning a value to it in the constructor
  - 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
  - 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;

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

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

- Like we use `this` when an instance calls a method on itself, we also use `this` when an instance references one of its instance variables after declaration.
  - Java compiler will work without it, but required in CS15 to easily distinguish instance variables from local variables.

- Thus, we use `this` to refer to capabilities (methods) and properties (instance variables) of an instance.

```java
public class PetShop {
    private DogGroomer groomer;
    public PetShop() {
        this.groomer = new DogGroomer();
        this.testGrooming();
    }
    public void testGrooming() {
        Dog django = new Dog();//local var
        this.groomer.trimFur(django);
    }
}
```
Instance Variables (3/4)

- 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
Instance Variables (4/4)

- 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 – very unsafe!

- 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;

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

    public void testGrooming() {
        Dog django = new Dog();// local var
        this.groomer.trimFur(django);
    }
}
```
Encapsulation Design Pattern

• Why *private* instance variables?

• **Encapsulation** for safety… your properties are your private business

• Allows for **chain of abstraction** so classes don’t need to worry about the inner workings of contained classes
  
  o we will also show you safe ways of allowing other classes to have selective access to designated properties… stay tuned
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;

    public PetShop() {
        // oops! Forgot to initialize groomer
        this.testGrooming();
    }

    public void testGrooming() {
        Dog django = new Dog(); // local var
        this.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 instance variables have been explicitly initialized, preferably in the constructor, and no variables are initialized as null.

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

    public PetShop() {
        //oops! Forgot to initialize groomer
        this.testGrooming();
    }

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

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

    public void testGrooming() {
        Dog django = new Dog(); //local var
        this.groomer.shampooFur(django);
        this.groomer.trimFur(django);
    }
}

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

public class DogGroomer {
    /* constructor elided */

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

    public void shampooFur(Dog dirtyDog) {
        //code that shampoos the fur of dirtyDog
    }
    ...
}
Visualizing Our PetShop Program
TopHat Question

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

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

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

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

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

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

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

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

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

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

A

App

Farm

House Pig Cow

B

App

Farm

House Pig Cow

C

App

House Cow

Pig Farm
Summary

- **containment**: when one instance is a component of another class so the container can therefore send messages to the component it created.

- **delegation pattern**: passing responsibility of task details to another class to maintain clean code design.
  - results in a chain of abstraction

- **local variables**: scope is limited to a method.

- **instance variables**: 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 variable, when the last reference to it is deleted.
Announcements

• Lab 1 (Intro to Java) begins today
  o Some section rooms assignments have changed, so be sure to read email from section TAs

• AndyBot due tomorrow 9/22
  o No late deadline = no credit for code submitted past the deadline

• If you feel like you could use extra practice writing code, attend workshops!
  o Check Ed for workshop schedule
Topics in Socially-Responsible Computing

Surveillance Capitalism
Surveillance Capitalism

• Market system based on gathering data via surveillance and using it not just to predict but also control consumer behavior
  ○ term coined by retired HBS Professor Shoshana Zuboff in 2014

Industrial capitalism relied on labor and land for the market dynamic

Surveillance capitalism translates private experience into commodities
  • Advertisers buy your information
Surveillance Capitalism

“I describe surveillance capitalism as the unilateral claiming of private human experience as free raw material for translation into behavioral data. These data are then computed and packaged as prediction products and sold into behavioral futures markets.”

– Shoshana Zuboff
Surveillance Capitalism, Broadly

- Companies like Google + Facebook rely on ads for revenue
  - Predict: more data collected → more precise + effective ads → $$$$  
  - Control: more time users spend on app → more ads served → $$$$  
- Behavior modification: surest way to predict behavior is to intervene at its source and shape it
  - Habit-forming apps get users to spend more time on them  
  - Expansion of surveillance, track users across apps  
  - Thousands of third party data brokers buy and sell YOUR data
    - 200 billion dollar industry
Use of addictive software design

- Utilize addictive tactics that stem from gambling

Surveillance capitalism has generated unbelievable wealth (and increased income inequality)
And market power...
Surveillance Capitalism in Action

- Cambridge Analytica: data firm owned by right-wing donor Robert Mercer
  - Data used by Steve Bannon to create voter profiles
  - Microtargeting lead to political bubbles
- Largest known misuse of data in Facebook history
  - Harvested data from over 87 million users
- Cambridge Analytica in contact with Lukoil—Kremlin linked oil giant
- 2018, Zuckerberg appear before Congress
  - Question of how data can/should be handled
  - Continues to appear and explain what Facebook is/isn’t doing
More reading that may be of interest!

- [The Age of Surveillance Capitalism](#) — Shoshana Zuboff
- [Cambridge Analytica and Facebook](#), NYTimes
- [Google Photos and Data Mining](#), Tech Crunch
- [Surveillance Capitalism and the Pandemic](#), M. Soules