

# Abstract Classes

# Brandon Krakowsky





# Abstract Classes



# Abstract Methods

- You can declare an Object without defining it:

```
Person p; //declares variable p
```

- Similarly, you can declare a method without defining it:

```
public abstract void draw(int size); //declares method draw
```

- Declares the method with the keyword *abstract*

- Notice that the body of the method is missing
  - Instead of curly braces { } you just have a semi-colon ending the statement
- A method that has been declared but not defined is an **abstract method**



# Abstract Classes

- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword *abstract*:

```
public abstract class Shape { //abstract class  
    public abstract void draw(int size); //abstract method WITHOUT body  
}
```

- You cannot instantiate (create a new instance of) an abstract class
  - You CANNOT do this:  
`Shape shape = new Shape();`



# Extending Abstract Classes

- You can (and usually do) extend (or subclass) an abstract class
  - If the subclass defines all of the inherited abstract methods, it is “complete” and can be instantiated
    - This is also known as a “concrete” class
  - If the subclass does not define all the inherited abstract methods, it too must be abstract
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated



## Example Abstract Class

- An abstract class can contain both *abstract* and *non abstract* (or concrete) methods
- Here's a sample abstract class for Pet
  - It contains an abstract method WITHOUT a body and a non abstract (concrete) method WITH a body

```
public abstract class Pet { //abstract class

    protected double weight;

    public abstract String makeSound(); //abstract method WITHOUT body

    public void eat(Food food) { //non abstract (concrete) method WITH body
        this.weight += (food.getCalories() / 100);
    }
}
```

- This class cannot be instantiated because it's abstract



## Example Abstract Class

- Subclasses of Pet must provide an implementation of the makeSound method

```
public class Dog extends Pet {  
  
    @Override  
    public String makeSound() { //implementation of abstract method  
        return "Bark!";  
    }  
  
    Dog myDog = new Dog();  
    System.out.println(myDog.makeSound()); //prints "Bark!"
```



# Why Have Abstract Classes?

- For this example, we know that all `Pets` eat
  - The effect on the food is always the same
  - Instead of implementing the `eat` method for each subclass of `Pet`, we can implement it once and inherit it
- On the other hand, different `Pets` make different sounds
  - We want to make sure that each subclass implements the `makeSound` method
- We don't want to instantiate a `Pet` object because we don't know what it is (it's abstract!)



## Common Syntax Error with Subclasses

- Suppose we have a Shape class

```
class Shape { ... }
```

- And things (“shapes”) that extend Shape

```
class Star extends Shape {  
    void draw() { ... }  
    ...  
}
```

```
class Crescent extends Shape {  
    void draw() { ... }  
    ...  
}
```



## Common Syntax Error with Subclasses

- Then we CAN actually do this, because a Star is a Shape

```
//declare variable of type Shape to store object of type Star  
Shape myShape = new Star();
```

- Unfortunately, we CANNOT do this

```
//trying to call draw on type Shape  
myShape.draw();
```

- This is a syntax error, because myShape (of type Shape) does not have a draw method
  - myShape doesn't know what the draw method is

- So how can we call draw on every thing (or “shape”)?

- Use an abstract class!



## Common Syntax Error with Subclasses

- Suppose we are making a GUI, and we want to draw a number of different “shapes” (circles, crescents, etc.)
  - Each class has a `draw` method
- You want to make the different shapes subclasses of `Shape`, so that you can create an `ArrayList<Shape>` `shapes` to hold the various things to be drawn:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();  
shapes.add(star); //add star to shapes  
shapes.add(crescent); //add crescent to shapes
```

- Then, you would like to do something like this:

```
for (Shape s : shapes)  
    s.draw(); //try to call draw on each type Shape
```

- Again, this is not legal, because each `s` (of type `Shape`) does not have a `draw` method
- Rule: Every class “knows” its superclass, but a class doesn’t “know” its subclasses
  - *You may know that every subclass of `Shape` has a `draw` method, but Java does not!*



## Possible Solutions to the Syntax Problem

- Solution 1: Put a `draw` method in the `Shape` class itself
  - This method will be inherited by all subclasses
  - Then you can call `draw` method for each shape
  - But what will it actually draw? It will be defined the same way for every subclass.
- Solution 2: Put an `abstract draw` method in the `Shape` class
  - This will also be inherited by all subclasses, but you won't have to define it in the `Shape` class
  - You do, however, have to make the `Shape` class abstract
  - And implement `draw` in each subclass
  - This way, Java knows that each subclass has a `draw` method



# Use Case for Abstract Class

- Make the Shape class abstract and add the abstract draw method with no body
  - Every subclass of Shape will HAVE TO implement draw

```
abstract class Shape {  
    abstract void draw();  
}  
  
class Star extends Shape {  
    @Override  
    void draw() { ... } //implement abstract draw method  
    ...  
}  
  
class Crescent extends Shape {  
    @Override  
    void draw() { ... } //implement abstract draw method  
    ...  
}  
  
Shape myShape = new Star();  
- This is legal, because a Star is a Shape  
- However, Shape myShape = new Shape(); is no longer legal  
  
myShape.draw(); //call draw method on type Shape
```



# Abstract Pet Project

# Pet Class



```
1 package pet;
2
3 import java.util.ArrayList;
4
5 /**
6  * Abstract class Pet to represent a generic (or abstract) pet.
7  * @author wcauser
8  *
9 */
10 public abstract class Pet {
11
12     //instance variables
13
14     /**
15      * Age of pet.
16      * Default (package-private) access. Accessible anywhere in same package.
17      */
18     int age;
19
20     /**
21      * Name of pet.
22      * Default (package-private) access. Accessible anywhere in same package.
23      */
24     String name;
25
26     /**
27      * Weight of pet.
28      * Default (package-private) access. Accessible anywhere in same package.
29      */
30     double weight;
31 }
```



# Pet Class

```
31     //constructor(s)
32
33
34⊕  /**
35   * Called by subclasses of Pet to create instances of different types of pets.
36   * @param name of pet
37   * @param age of pet
38   * @param weight of pet
39   */
40⊕  public Pet(String name, int age, double weight) {
41      this.name = name;
42      this.age = age;
43      this.weight = weight;
44  }
45
46  //getters and setters
47
48⊕  /**
49   * @return the weight
50   */
51⊕  public double getWeight() {
52      return this.weight;
53  }
54
55⊕  /**
56   * @param weight the weight to set
57   */
58⊕  public void setWeight(double weight) {
59      this.weight = weight;
60  }
61
```

# Pet Class

```
51  
62     //non-abstract method(s)  
63  
64 $\ominus$   /**
65      * Tells pet to eat given food.
66      * @param food to eat
67      */
68 $\ominus$   public void eat(Food food) {
69      this.weight += (food.getCalories() / 100);
70  }
71
```

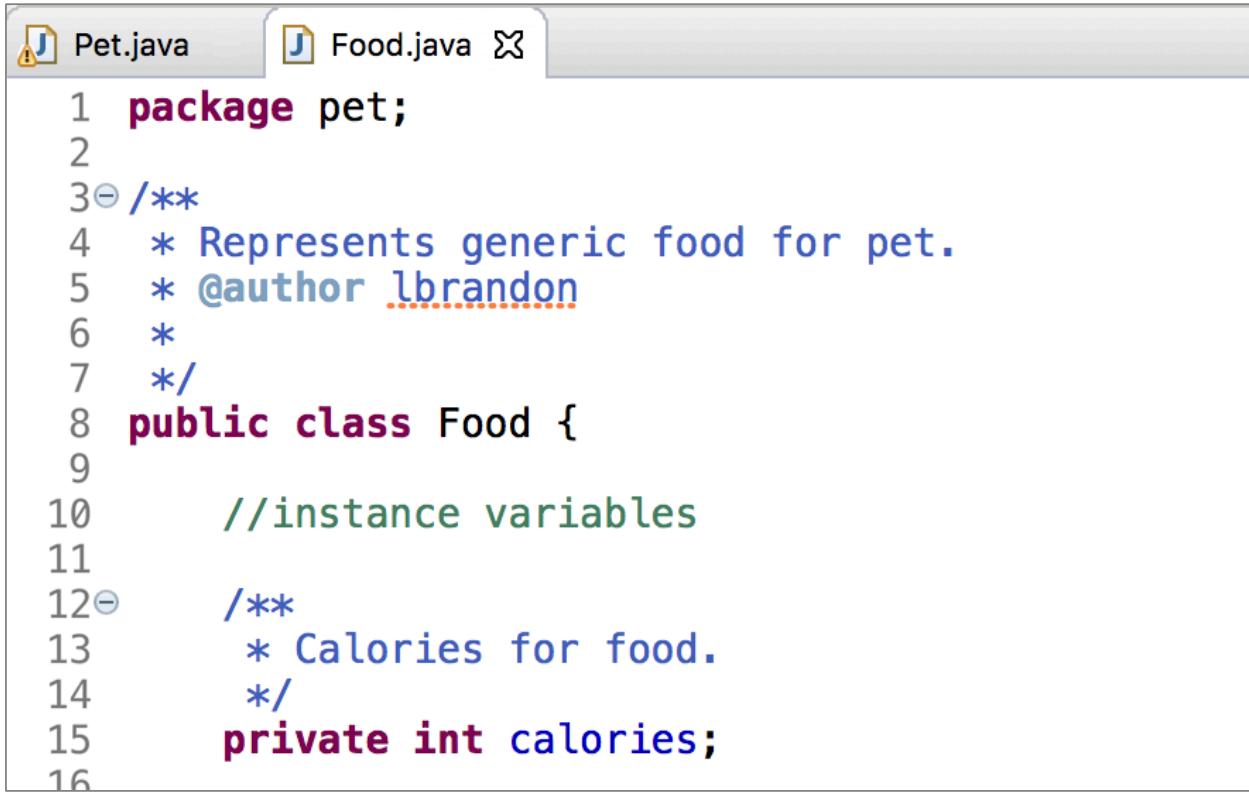


# Pet Class

```
71
72     //abstract method(s)
73
74     /**
75      * Forces all pets to make their own sound.
76      * All subclasses of Pet MUST implement this method.
77      */
78     public abstract void makeSound();
79
80     /**
81      * Every subclass must implement toString for printing/debugging.
82      */
83     @Override
84     public abstract String toString();
85
```



# Food Class



```
1 package pet;
2
3 /**
4  * Represents generic food for pet.
5  * @author lbrandon
6  *
7 */
8 public class Food {
9
10    //instance variables
11
12 /**
13  * Calories for food.
14  */
15 private int calories;
16}
```

# Food Class

```
16
17     //constructor(s)
18
19⊕    /**
20     * Creates instance of Food.
21     * @param calories for food
22     */
23⊕    public Food(int calories) {
24        this.calories = calories;
25    }
26
27    //getters/setters
28
29⊕    /**
30     * @return the calories
31     */
32⊕    public int getCalories() {
33        return calories;
34    }
35
36⊕    /**
37     * @param calories the calories to set
38     */
39⊕    public void setCalories(int calories) {
40        this.calories = calories;
41    }
42
```



# Cat Class

```
1 package pet;
2
3 /**
4 * Represents a Cat, subclass of abstract class Pet.
5 * @author lbrandon
6 *
7 */
8 public class Cat extends Pet {
9
10    //static variables
11
12 /**
13 * Default sound for all cats.
14 * Private access. Only accessible within this class.
15 */
16 private static String SOUND = "Meow!";
17
18 //instance variables
19
20 /**
21 * Type of this cat.
22 * Private access. Only accessible within this class.
23 */
24 private String type;
25}
```



# Cat Class

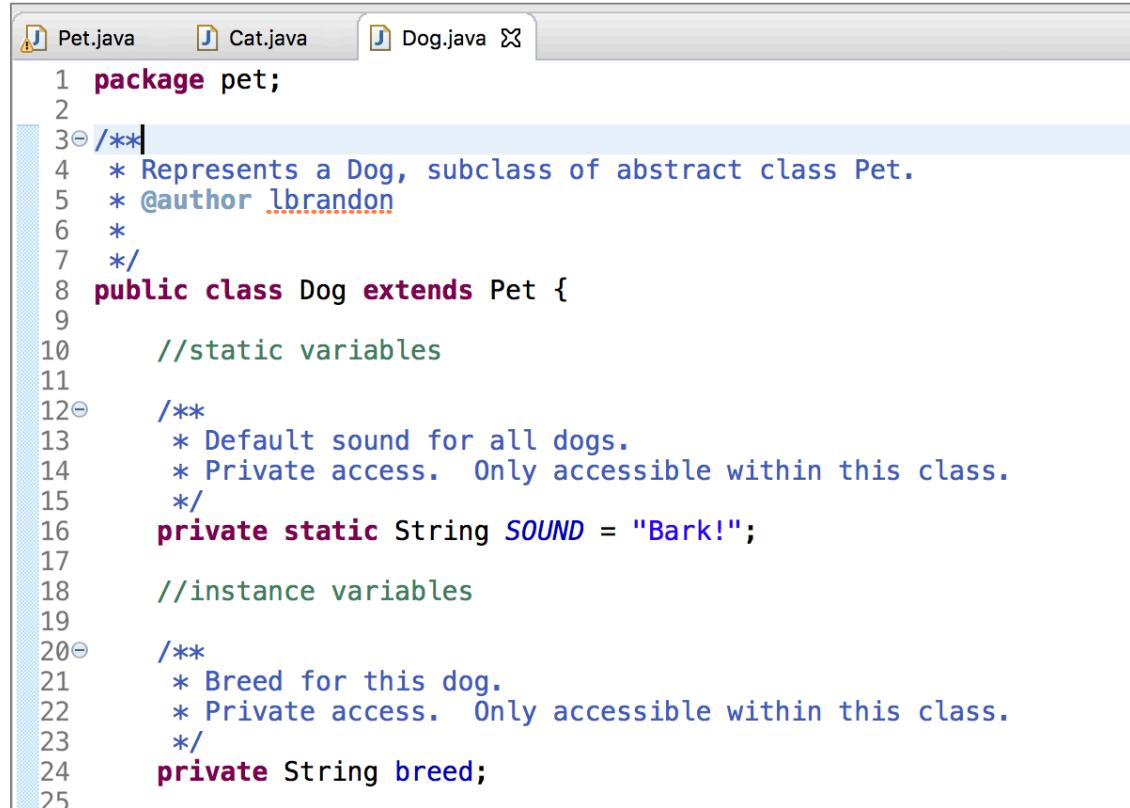
```
26    //constructor(s)
27
28    /**
29     * Creates cat with given name, age, weight, and type.
30     * @param name for cat
31     * @param age for cat
32     * @param weight for cat
33     * @param type for cat
34     */
35    public Cat(String name, int age, double weight, String type) {
36        //must call constructor in superclass Pet
37        super(name, age, weight);
38
39        //set type
40        this.type = type;
41    }
42
43    //getters/setters
44
45    /**
46     * @return the type
47     */
48    public String getType() {
49        return type;
50    }
51
52    /**
53     * @param type the type to set
54     */
55    public void setType(String type) {
56        this.type = type;
57    }
```



## Cat Class

```
59     //inherited abstract methods
60     //MUST override and implement these (provide bodies)
61
62     /**
63      * Makes cat sound.
64      */
65     @Override
66     public void makeSound() {
67         System.out.println(Cat.SOUND);
68     }
69
70     /**
71      * Returns string for printing/debugging this cat.
72      */
73     @Override
74     public String toString() {
75         return this.name + " is a " + this.type;
76     }
77 }
```

# Dog Class



```
1 package pet;
2
3 /**
4 * Represents a Dog, subclass of abstract class Pet.
5 * @author lbrandon
6 *
7 */
8 public class Dog extends Pet {
9
10     //static variables
11
12 /**
13 * Default sound for all dogs.
14 * Private access. Only accessible within this class.
15 */
16 private static String SOUND = "Bark!";
17
18     //instance variables
19
20 /**
21 * Breed for this dog.
22 * Private access. Only accessible within this class.
23 */
24 private String breed;
25 }
```

# Dog Class

```
27
28⊕  /**
29   * Creates dog with given name, age, weight, and breed.
30   * @param name of dog
31   * @param age of dog
32   * @param weight of dog
33   * @param breed of dog
34   */
35⊕ public Dog(String name, int age, double weight, String breed) {
36     //must call constructor in superclass Pet
37     super(name, age, weight);
38
39     //set breed
40     this.breed = breed;
41 }
42
43 //getters/setters
44
45⊕ /**
46   * @return the breed
47   */
48⊕ public String getBreed() {
49     return breed;
50 }
51
52⊕ /**
53   * @param breed the breed to set
54   */
55⊕ public void setBreed(String breed) {
56     this.breed = breed;
57 }
```



# Dog Class

```
58  
59     //inherited abstract methods  
60     //MUST override and implement these (provide bodies)  
61  
62⊕    /**  
63     * Makes dog sound.  
64     */  
65⊖    @Override  
66    public void makeSound() {  
67        System.out.println(Dog.SOUND);  
68    }  
69  
70⊖    /**  
71     * Returns string for printing/debugging this dog.  
72     */  
73⊖    @Override  
74    public String toString() {  
75        return this.name + " is a " + this.breed;  
76    }  
77
```

# Pet Class

```
86
87  public static void main(String[] args) {
88
89      //create list of pets
90      ArrayList<Pet> pets = new ArrayList<Pet>();
91
92      //create a dog
93      Dog dog = new Dog("Buster", 14, 100, "German Shepherd");
94
95      //create a cat
96      Cat cat = new Cat("Snuggles", 3, 40, "House Cat");
97
98      //add pets to list
99      pets.add(dog);
100     pets.add(cat);
101
```



# Pet Class

```
101
102     //iterate over list of pets
103     for (Pet p : pets) {
104         //call makeSound, abstract method in pet class
105         //every subclass of pet MUST have this defined
106         p.makeSound();
107     }
108
109     //iterate over list of pets
110     for (Pet p : pets) {
111         //print each pet
112         //calls toString method implemented in each subclass of pet
113         System.out.println(p);
114     }
115
116     //call specific method in dog class
117     String breed = dog.getBreed();
118     System.out.println("dog breed: " + breed);
119
120     //call specific method in cat class
121     String type = cat.getType();
122     System.out.println("cat type: " + type);
123
```

# Pet Class

```
23
24     //get dog weight before eating
25     //getWeight defined as non-abstract method in pet class
26     System.out.println(dog.getWeight());
27
28     //create some food
29     Food food = new Food(1000);
30
31     //dog will eat
32     //eat defined as non-abstract method in pet class
33     dog.eat(food);
34
35     //get dog weight after eating
36     System.out.println(dog.getWeight());
37
```

