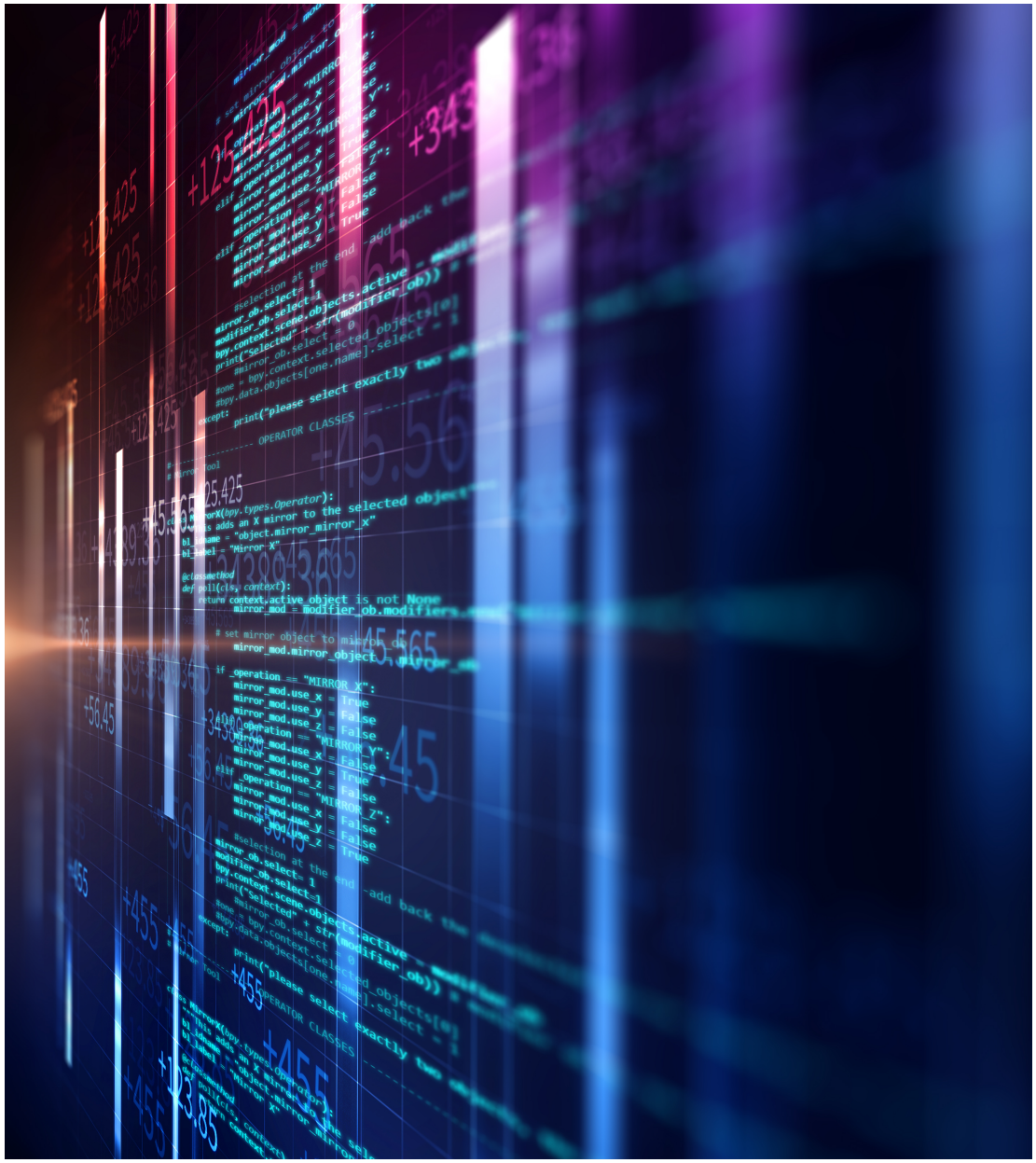
Polymorphism – Inheritance & Overriding

Brandon Krakowsky



Penn Engineering

Inheritance

Inheritance

- Inheritance is the mechanism by which one class *inherits* the fields and methods of another class
- The class whose features are inherited is known as the superclass (or parent class)
- The class that inherits the other class is known as the subclass (or extended class, or child class)
- The subclass can add its own fields and methods in addition to the superclass fields and methods
- Inheritance allows “reusability”: deriving new classes from existing classes that already have code (fields and methods) that we want to use

Inheritance

```
public class Animal {

    public void greeting() {
        System.out.println(”I am an animal!");
    }
}

public class Dog extends Animal {

    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();


        //prints “I am an animal!”
        animal.greeting();

        //prints “I am an animal!”
        dog.greeting();
    }
}
```

- Animal is the *superclass*
- Dog is the *subclass*
- Dog *extends* Animal
- Dog *inherits* greeting() method from Animal

Overriding

Review of Method Signature

In any programming language, a signature is what distinguishes one function or method from another

- Remember, in Java, a method signature is defined by its *name*, the *number* of its parameters, the *types* of its parameters, and the *sequence* of its parameters
- Examples:
  - foo(int i) and foo(int i, int j) are considered different signatures
  - foo(int i) and foo(int k) are considered the same signatures
  - foo(int i, double d) and foo(double d, int i) are considered different signatures
- A method signature does not include the return type

How to Override a Method

- Create a method in a subclass having the same *signature* as a method in a superclass
- That is, create a method in a subclass having the same *name* and the same *number*, *types*, and *sequence* of parameters
  - Parameter *names* don't matter, just their *types*
- Restrictions:
  - The return type of the method must be the same
  - The overriding method cannot be *more private* than the method it overrides (ignore this bullet point for now)

How to Override a Method

```
public class Animal {

    public void greeting() {
        System.out.println(”I am an animal!");
    }
}
public class Dog extends Animal {

    @Override
    public void greeting() {
        System.out.println(”I am a dog!");
    }

    public static void main(String args[]) {
        Animal animal = new Animal();
        Dog dog = new Dog();

        //prints “I am an animal!”
        animal.greeting();

        //prints “I am a dog!”
        dog.greeting();
    }
}
```

- Animal is the *superclass*
- Dog is the *subclass*
- Dog *extends* Animal
- Method greeting() in Dog *overrides* method greeting() in Animal
- The @Override annotation informs the compiler (and those using your code) that the element is meant to override an element declared in a superclass
  - It’s not required, but helps to prevent errors

Why Override a Method?

- If you want the method to do something different than in the superclass
- For example, if you try to print a Dog, you don't get what you expect (or probably want)

```
Dog dog = new Dog();
System.out.println(dog);
```

  - This will print something like Dog@feda4c00
- How does this work?
  - The *System.out.println* method calls the *toString()* method, which is defined in Java's top-level Object class
    - Hence, every object can be printed (though it might not look pretty)
  - Every class (including Dog) is a subclass of Object and inherits the *toString()* method
    - Hence, you can override the *toString()* method in any class (including Dog)

A Review of Overriding *toString*

- To override the public String toString() method in the Dog class

```
public class Dog extends Animal {

    String name;

    @Override
    public String toString() {
        return this.name;
    }
}
```

- Then System.out.println(dog); will print the dog's name
  - This probably makes more sense for our purposes!

- Reminder: To override a method, create a method with the same *name,* and the same *number*, *types*, and *sequence* of *parameters*
  - The return type of the method must be the same

More About *toString*

It is almost always a good idea to override *toString()* in a class and to return something "meaningful" about the object

- When debugging, it helps to be able to print objects
- When you print objects with *System.out.println* (or *System.out.print*), they automatically call the object's *toString()* method

  Example:
  ```
  System.out.println(dog);
  ```
- When you concatenate the object with another String, the object's *toString()* method is also automatically called

  Example:
  ```
  String info = "Info about dog:" + dog;
  System.out.println(info);
  ```

A Review of Equality

- Consider these two animals:

```
Animal animal1 = new Animal();
Animal animal2 = new Animal();
```

- Are they equal? What does it mean for them to be equal?
  - This is up to the programmer!

A Review of Overriding *equals*

Object equality is tested with the public boolean equals(Object o) method

- Unless it's overridden, this method just uses == (typically only used for comparing primitive values)
- IT IS overridden in the String class
  - For example, this is true if the 2 Strings have the same value

```
String string1 = "thisString";
String string2 = "thisString";
string1.equals(string2); //true
```

- IT IS NOT overridden for arrays (defaults to ==)
  - For example, this is true only if the 2 arrays are the same object

```
int[] array1 = {1, 2, 3};
int[] array2 = {1, 2, 3};
array1.equals(array2); //false
```

  - This is why we use the Arrays.equals(array1, array2) method to compare array values

A Review of Overriding *equals*

- To test for object equality in your own classes, override the public boolean equals(Object o) method

```
public class Animal {

    String name;

    @Override
    public boolean equals(Object o) {

        //cast o to Animal
        Animal otherAnimal = (Animal) o;

        //compare this.name to name of the animal passed in
        return this.name.equals(otherAnimal.name);
    }
}
```

A Review of Overriding *equals* for JUnit Tests

- Why is the *equals* method important for JUnit testing?
- By default, the JUnit method *assertEquals(expected, actual)* uses == to compare *primitives* and *equals* to compare *objects*
- This means that if you want to use *assertEquals* with your own objects, YOU MUST implement the *equals* method in your class
- To do that, define this method (exactly) in your class:

  public boolean equals(Object *obj*) { ... }
  - The argument must be of type Object, which isn't what you want, so you must cast it to the correct type (e.g. Person):

Overriding *equals* in the Fraction Class

- For example, in the Fraction assignment, you have to correctly implement the *equals* method in your Fraction class

```
class Fraction {

    @Override
    public boolean equals(Object obj) {

        //create new fraction object and reduce to lowest form
        Fraction thisFraction = new Fraction(this.numerator, this.denominator);
        thisFraction.reduceToLowestForm();

        //cast given obj to Fraction and reduce to lowest form
        Fraction otherFraction = (Fraction) obj;
        otherFraction.reduceToLowestForm();

        //write code to compare thisFraction to otherFraction
    }
}
```

Comparing Fractions in the Fraction Testing Class

- You can then compare two Fraction objects in your Fraction testing class using *assertEquals*

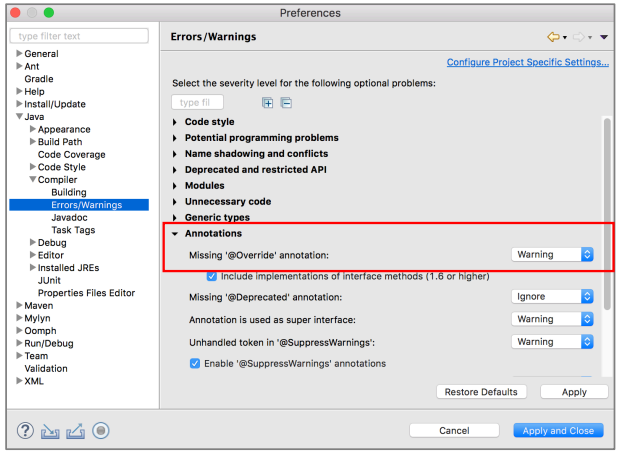
```
class FractionTest {

    @Test
    void testEquals() {
        Fraction fraction1 = new Fraction(2, 3);
        Fraction fraction2 = new Fraction(2, 3);
        assertEquals(fraction1, fraction2);

        fraction1 = new Fraction(4, 16);
        fraction2 = new Fraction(1, 4);
        assertEquals(fraction1, fraction2);

        //write more test cases
    }
}
```

More About the @Override Annotation

How to “Require” @Override

When overriding a method, the @Override annotation is not required in Java

- The default preference in Eclipse is to ignore any warnings about missing annotations, but it’s a good idea to change this preference for @Override
  - Go to Eclipse --> Preferences
  - Then go to Java --> Compiler --> Errors/Warnings, and change the 'Missing @Override annotation' setting to "Warning"



Overriding Constructors

Overriding Constructors - I

- The very first thing any constructor does, automatically, is call the *default* constructor in its superclass

```
public class Foo extends Bar {

    public Foo() { // constructor
        super(); // invisible call to constructor in superclass (Bar)
        ...
    }
}
```

Overriding Constructors - II

- You can replace this with a call to a *specific* constructor *in the superclass*
  - Use the keyword super
  - This must be the *very first thing* the constructor does

```
class Foo extends Bar {

    public Foo(String name) { // constructor
        super(name); // explicit call to specific constructor in
superclass (Bar)
        ...
    }
}
```

Overriding Constructors - III

- You can replace this with a call to a *specific* constructor *in this class*
  - Use the keyword this
  - This must be the *very first thing* the constructor does

```
class Foo extends Bar {

    public Foo(String message) { // constructor
        this(message, 0, 0); // explicit call to another constructor
in this class (Foo)
        ...
    }
}
```

Calling an Overridden Method

- When your class overrides an inherited method, it basically “hides” the inherited method
- Within this class (but not from a different class), you can still call the overridden method, by prefixing the call with super
- Here, we call the *greeting* method in the Animal class

```
public class Dog extends Animal {

    @Override
    public void greeting() {
        super.greeting(); //call overridden greeting
        //method in superclass (Animal)
        System.out.println(”I am a dog!");
    }

    public static void main(String args[]) {
        Dog dog = new Dog();
        dog.greeting();
        //prints:
        //“I am an animal!”
        //”I am a dog!”
    }
}
```

Calling an Overridden Method

- When your class overrides an inherited method, it basically “hides” the inherited method
- Within this class (but not from a different class), you can still call the overridden method, by prefixing the call with super
- Here, we call the *greeting* method in the Animal class

```
public class Dog extends Animal {

    @Override
    public void greeting() {
        System.out.println(”I am a dog!");
        super.greeting(); //call overridden greeting
        //method in superclass (Animal)
    }

    public static void main(String args[]) {
        Dog dog = new Dog();
        dog.greeting();
        //prints:
        //“I am a dog!”
        //”I am an animal!”
    }
}
```

- Since this isn’t a call to a constructor in the superclass, it can occur anywhere in your method (or class)
  - It doesn’t have to be first

Why Call an Overridden Method?

- You would most likely call an overridden method in order to observe the DRY (Don't Repeat Yourself) principle of software development
  - The superclass method will do most of the work, but you want to add to it or adjust its results

Animals Management Project

Animals Management Project

- This project will represent a manager with different kinds of animals
- The Animal class will represent a generic animal
- The Dog class will extend Animal, and represent a dog
- The Cat class will extend Animal, and represent a cat
- The AnimalsManager class will drive the program and create various kinds of animals

Animal Class

Animal.java

```java
package animal;

/**
 * Represents a generic animal.
 * We will create animal sub-classes (dog, cat, etc.) that inherit from this class.
 * @author lbrandon
 */
public class Animal {

    //static variables

    /**
     * Default age for animal.
     */
    private static int DEFAULT_AGE = 0;


    //instance variables

    /**
     * Age of animal.
     */
    int age;

    /**
     * Weight of animal.
     */
    double weight;

    /**
     * Name of animal.
     */
    String name;
```

Animal Class

```
35      //constructor(s)
36
37      /**
38       * Create animal with given age.
39       * @param age of animal
40       */
41      public Animal(int age) {
42          this.age = age;
43      }
44
45      /**
46       * Create animal with default age.
47       */
48      public Animal() {
49          //call another constructor in this class
50          this(Animal.DEFAULT_AGE);
51      }
```

Animal Class

```
53      //getters/setters
54
55      /**
56       * @return the name
57       */
58      public String getName() {
59          return name;
60      }
61
62      /**
63       * @param name the name to set
64       */
65      public void setName(String name) {
66          this.name = name;
67      }
68
```

Animal Class

```
69      //other methods
70
71      /**
72       * Animal speaks.
73       */
74      public void speak() {
75          System.out.println("Animal says hi.");
76      }
77
78      /**
79       * Overrides toString in Object class.  (Every class is a subclass of Object.)
80       * Returns name of animal for printing.
81       */
82      @Override
83      public String toString() {
84          return this.name;
85      }
```

Cat Class

Animal.java | Cat.java

```
1  package animal;
2
3  /**
4   * Represents a cat and extends Animal.
5   * @author lbrandon
6   *
7   */
8  public class Cat extends Animal {
9
10     //static variables
11
12     /**
13      * Default type for cat.
14      */
15     private static String DEFAULT_TYPE = "domestic";
16
17
18     //instance variables
19
20     /**
21      * Type of cat.
22      */
23     private String type;
24
25     //Note: Also inherits non-private variables defined in superclass (Animal)
```

Cat Class

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

Cat Class

```
56
57      //other methods
58
59      /**
60       * Overrides speak method in superclass (Animal)
61       * Cat speaks.
62       */
63      @Override
64      public void speak() {
65          System.out.println(this.name + " says: Meow!");
66      }
67
68      /**
69       * Overrides toString method in superclass (Animal)
70       * Returns name and type for cat.
71       */
72      @Override
73      public String toString() {
74          return this.name + " is a " + this.type;
75      }
```

Dog Class

Animal.java | Cat.java | Dog.java

```
1  package animal;
2
3  /**
4   * Represents a dog and extends Animal.
5   * @author lbrandon
6   *
7   */
8  public class Dog extends Animal {
9
10     //instance variables
11
12     /**
13      * Breed for dog.
14      */
15     private String breed;
16
17     //Note: Also inherits non-private variables defined in superclass (Animal)
```

Dog Class

```
18
19      //constructor(s)
20
21      /**
22       * Create a dog with given age and breed.
23       * @param age of dog
24       * @param breed of dog
25       */
26      public Dog(int age, String breed) {
27          //call constructor in superclass Animal
28          super(age);
29
30          //set breed
31          this.breed = breed;
32
33      }
34
35      //getters/setters
36
37      /**
38       * @return the breed
39       */
40      public String getBreed() {
41          return breed;
42      }
43
44      /**
45       * @param breed the breed to set
46       */
47      public void setBreed(String breed) {
48          this.breed = breed;
49      }
50
```

Dog Class

```
51      //other methods
52
53      /**
54       * Overrides speak method in superclass (Animal)
55       * Dog speaks.
56       */
57      @Override
58      public void speak() {
59          System.out.println(this.name + " says: fear my bark!");
60
61          //calls speak method in superclass (animal)
62          super.speak();
63      }
64
65      /**
66       * Overrides toString method in superclass (Animal)
67       * Returns name and breed for dog.
68       */
69      @Override
70      public String toString() {
71          return this.name + " is a " + this.breed;
72      }
```

Dog Class

```
73
74      /**
75       * Overrides equals method in superclass (Object).  (Every class is a subclass of Object.)
76       * Compares two dogs, which are considered equal if they have the same name and breed.
77       */
78      @Override
79      public boolean equals(Object o) {
80
81          //cast to dog
82          Dog otherDog = (Dog) o;
83
84          //compare two dogs based on name and breed
85          return (this.name.equals(otherDog.name) && this.breed.equals(otherDog.breed));
86      }
87
```

DogTest Class

Animal.java | Cat.java | Dog.java | DogTest.java | AnimalsManager.java

```
1  package animal;
2
3  import static org.junit.jupiter.api.Assertions.*;
7
8  class DogTest {
9
10     @BeforeEach
11     void setUp() throws Exception {
12
13     }
14
15     @Test
16     void testEquals() {
17
18         //assumes we've implemented equals in the Dog class
19
20         Dog dog1 = new Dog(4, "Pug");
21         dog1.setName("Tommy");
22
23         Dog dog2 = new Dog(4, "Pug");
24         dog2.setName("Tommy");
25
26         //dogs should be equal
27         assertEquals(dog1, dog2);
28
29     }
30 }
```

AnimalsManager Class

Animal.java | Cat.java | Dog.java | AnimalsManager.java

```
1  package animal;
2  import java.util.ArrayList;
3
4  /**
5   * A class for managing animals.
6   * @author lbrandon
7   *
8   */
9  public class AnimalsManager {
10
11     //instance variables
12
13     /**
14      * List of all animals.
15      */
16     private ArrayList<Animal> animals;
17
18     /**
19      * Creates instance of AnimalsManager and initializes animal list.
20      */
21     public AnimalsManager() {
22         this.animals = new ArrayList<Animal>();
23     }
```

AnimalsManager Class

```
24
25      //methods
26
27      /**
28       * Prints all animals.
29       */
30      public void printAnimals() {
31          //iterates over animals and prints each one
32          for (Animal a : this.animals) {
33              System.out.println(a);
34          }
35      }
36
37      /**
38       * Tells all animals to speak.
39       */
40      public void animalsSpeak() {
41          //iterates over animals and tells each one to speak
42          for (Animal a : this.animals) {
43              a.speak();
44          }
45      }
46
```

AnimalsManager Class

```
46
47    public static void main(String[] args) {
48
49        //create generic animal
50        Animal animal1 = new Animal(2);
51
52        //set name for animal
53        animal1.setName("Bob the animal"); //calls setName method defined in Animal
54
55        System.out.println(animal1); //calls toString method in animal class
56
57        //create a dog
58        Dog dog1 = new Dog(4, "Pug");
59        dog1.setName("Puggles"); //calls setName method defined in Animal
60
61        //create another dog
62        Dog dog2 = new Dog(9, "Pug");
63        dog2.setName("Puggles"); //calls setName method defined in Animal
64
65        //create a cat
66        Cat cat1 = new Cat(8);
67        cat1.setName("Teddy"); //calls setName method defined in Animal
68        cat1.setType("outside"); //calls setType method defined in Cat
69
70        System.out.println(cat1); //calls toString method in cat class
71
```

AnimalsManager Class

```
71
72          //create instance of AnimalsManager
73          AnimalsManager animalManager = new AnimalsManager();
74
75          //get reference to arraylist animals
76          ArrayList<Animal> animalsList = animalManager.animals;
77
78          //add each animal to arraylist
79          animalsList.add(animal1);
80          animalsList.add(dog1);
81          animalsList.add(dog2);
82          animalsList.add(cat1);
83
84          //print all animals
85          animalManager.printAnimals(); //calls tostring method in each class
86
87          //tell all animals to speak
88          animalManager.animalsSpeak(); //calls speak method in each class
89
90          //compare dog1 to dog2.  are they equal?
91          //calls equal method in dog class
92          System.out.println("dog1.equals(dog2): " + dog1.equals(dog2));
93
```

Summary: Overloading vs. Overriding

- Overload Rule: You should *overload* a method when you want to do essentially the same thing, but with different parameters
- Override Rule: You should *override* an inherited method if you want to do something different than in the superclass
  - It's almost always a good idea to override public void toString() -- it's handy for debugging, and for many other reasons
  - To test your own objects for equality, override public void equals(Object o)