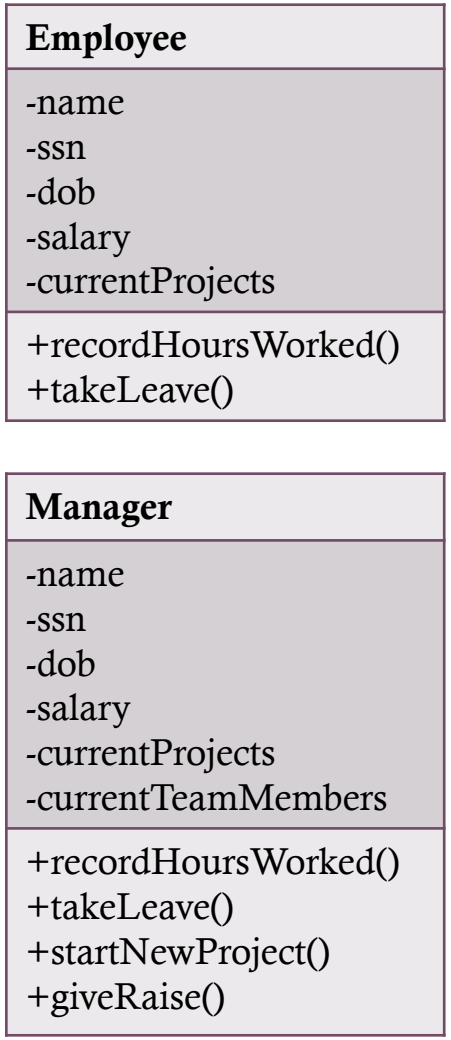
WEEK ONE

Acknowledgements: Slides created based off material provided by Dr. Travis Doom and Dr. Michael Raymer

INHERITANCE

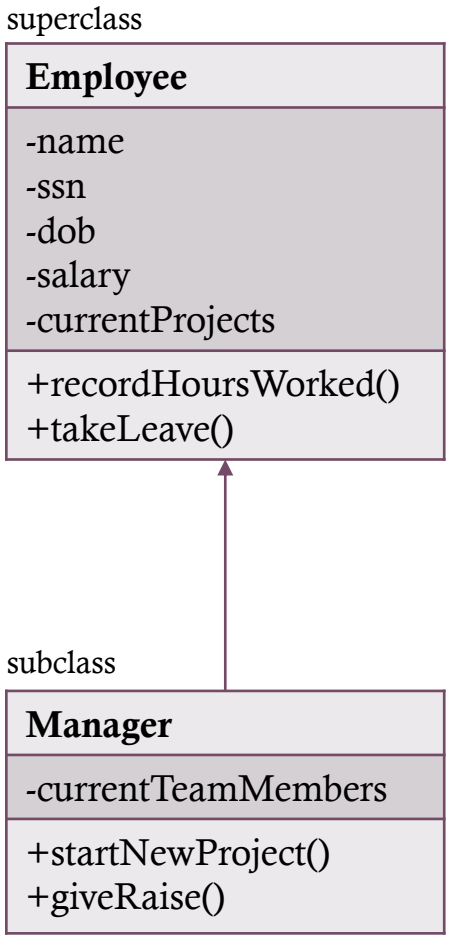
- The idea that separate classes can share information and functionality
- Helps avoid code duplication
- This is accomplished by one class *extending* another
  - The child (subclass) extends the parent (superclass)
  - This provides the child with access to all the fields and methods of the parent class



INHERITANCE

- Manager **IS-A** Employee
  - Employee is a more generic class than Manager
  - Manager **inherits** functionality from Employee
- Manager can have additional fields/methods

```
public class Manager extends Employee {
        …
        public Manager() {
        }
        public void startNewProject(String name) {
        }
        …
}
```



OVERRIDING

- Remember, overloading occurs when we have multiple methods with the same name but different parameters in the same class
  - `public void randomMethod(int a)`
  - `public void randomMethod(double a)`
- Overriding occurs when we have two methods with the exact same name and parameters in different classes (specifically in a parent and child class)
  - Allows us to change the functionality of a method for a specific child class
  - For example, the `takeLeave()` method may have different checks to perform for an Employee vs. a Manager before determining if that person can take leave

SUPER KEYWORD

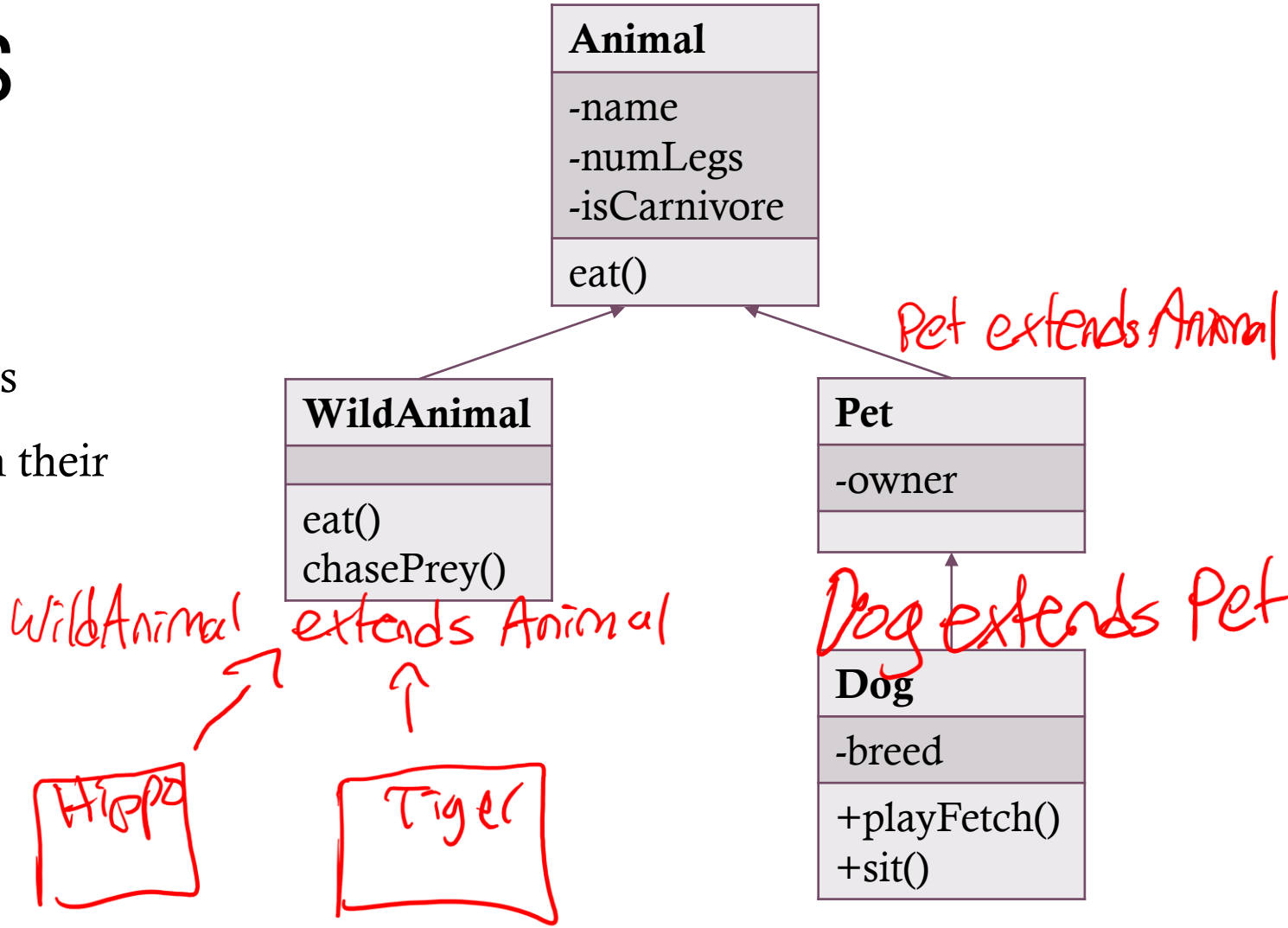
- Remember, the `this` keyword refers to the current class
- The `super` keyword refers to the parent class
- `super` can be used to call the parent class constructor
  - Good style to do this explicitly in every subclass constructor
  - `super()`
- `super` can also be used to call methods or access (public or protected) variables from the parent class
  - `super.recordHoursWorked();`
  - `super.publicParentClassField;`

PROTECTED ACCESS MODIFIER

- Allows the subclass full access to the variable/method
- ALSO allows all classes in the package full access to the variable/method
  - Not ideal
- Indicated by # in UML
- Utilize super, and getters/setters to avoid access control issues
  - `super.setName("Bob");`
  - In this case, name could have private access

INHERITANCE TREES

- Subclasses can also be extended
- Allows for multi-layer inheritance trees
- Objects call the "lowest" method from their start in the tree
- Dog **IS-A** Pet and **IS-A** Animal
- WildAnimal **IS-A** Animal
- Pet **HAS-A** owner
- Dog **HAS-A** breed



THE OBJECT CLASS

- If a class does not explicitly extend another class, then it implicitly extends Object
  - Thus, every class eventually inherits from the Object class
  - It is the mother/superclass of all objects
- Features the default constructor and default methods
  - `equals(Object o), toString(), clone(Object o), etc.`
- Allows us to create methods or routines that work for all classes
  - Remember we can create an ArrayList of Object to hold different objects.

INHERITANCE STYLE TIPS

- Look for classes that have common attributes/behaviors
- Design a class that represents that common state/behavior
- Determine if the subclasses need their own specific behaviors (methods)
- Look for any opportunities for abstraction
  - Two or more classes that share the same common behavior
  - Avoid duplicated code
- Remember, inheritance works in one direction
  - Superclass cannot inherit anything from subclasses

INHERITANCE SUMMARY

- A subclass extends a superclass
- A subclass inherits all *public* instance variables and methods of the superclass, but does not inherit the *private* instance variables and methods of the superclass
- Inherited methods can be overridden; instance variables cannot be overridden
- Use the IS-A test to verify your inheritance hierarchy. If X extends Y then X IS-A Y must make sense
- The IS-A relationship only works in one direction. A penguin IS-A bird, but not all birds are penguins.
- When a method is overridden in a subclass, and that method is invoked on an instance of the subclass, the overridden version of the method is called. (The lowest one wins.)
- Inheritance is transitive. If class B extends class A, and C extends B, class B IS-A class A and class C IS-A class B, and class C also IS-A class A.
- You get rid of a lot of duplicated code by using inheritance. If you need to change the shared behavior, you just have to update it in the superclass. All classes that extend it will automatically use the new version.

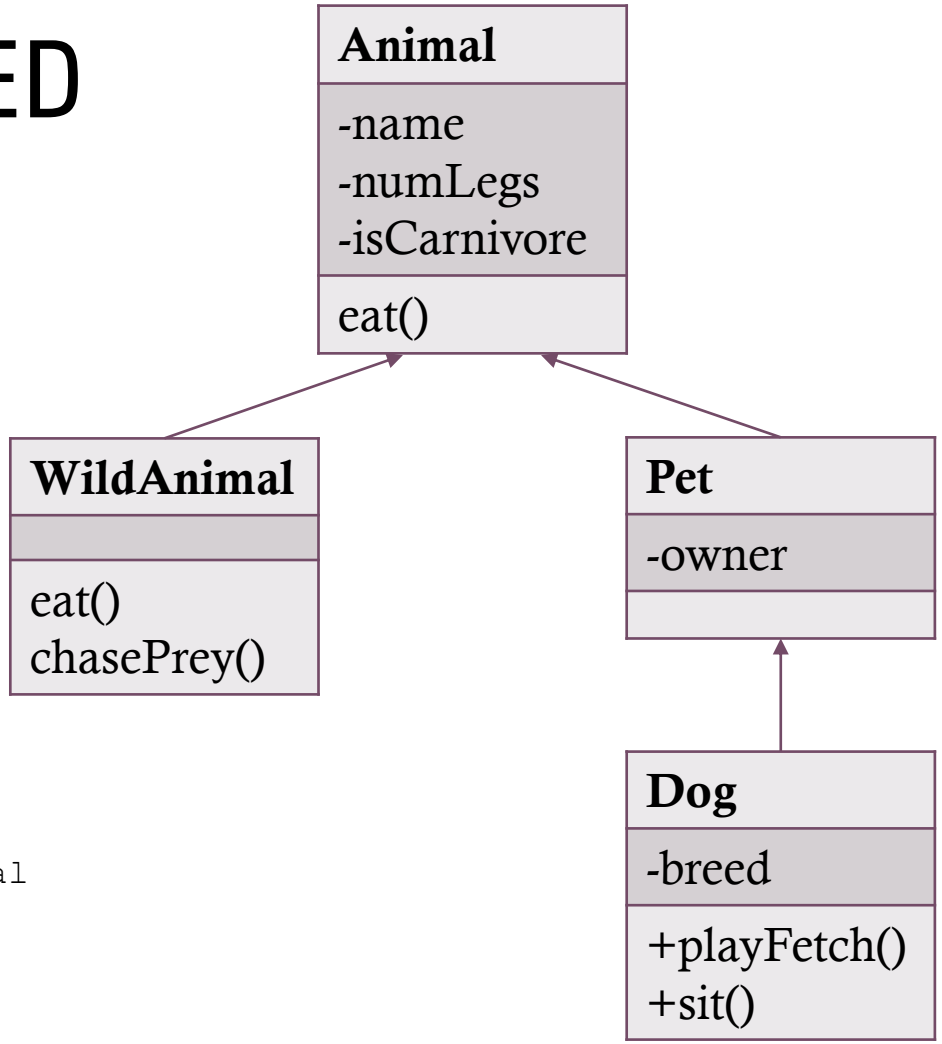
POLYMORPHISM

- The idea that all subclasses of a superclass can be represented by the superclass type
  - `Animal a = new WildAnimal();`
- An instance of any class (object) is considered to match type with all of its superclasses
- Any object that passes the IS-A test with any class can be stored/passed/returned as a reference of that type

```
Animal[] animalList = new Animal[5];
animalList[0] = new WildAnimal();
animalList[1] = new Animal();
animalList[2] = new Dog();
animalList[3] = new Pet();
```

POLYMORPHISM CONTINUED

```
public class Trainer {
        …
        public void feed(Animal a) {
                a.eat();
}}
public class Main {
        public static void main(String[] args) {
                Trainer t = new Trainer()
                WildAnimal tiger = new WildAnimal();
                Dog puppy = new Dog();
                t.feed(tiger); //calls eat() in WildAnimal
                t.feed(puppy); //calls eat() in Animal
}}
```



POLYMORPHISM SUMMARY

- Methods with polymorphic arguments (parameter is declared as superclass type) can take in any subclass at runtime
- Any subclass (no matter when it is added or by who) will also be able to be passed into the method
- Extending an existing class means that our subclass can be used wherever the superclass is contracted

INSTANCEOF OPERATOR

- How can we tell exactly what class a polymorphic object is an instance of?
- The `instanceof` operator can be used to check what specific class an object is an instance of

```
Animal[] animalList = new Animal[5];
animalList[0] = new WildAnimal();
animalList[3] = new Pet();
int numWildAnimals = 0;
int numPets = 0;
for (Animal a : animalList) {
        if (a instanceof WildAnimal) {
                numWildAnimals++;
        } else if (a instanceof Pet) {
                numPets++;
}}
```

DEBUGGING & EXCEPTION HANDLING EXAMPLE

REMEMBER COPY CONSTRUCTORS?

- A constructor that has an object of the same class as a parameter
- Makes an identical copy or clone of the object

```
public class Course {
        private String name = “”;
        private int creditHours = 0;

        public Course(Course originalCourse) {
                this.setName(originalCourse.getName());
                this.setCreditHours(originalCourse.getCreditHours());
        }
}
```

SHALLOW COPY

```
public class Student {
        private ArrayList<Course> classes = new ArrayList<>();

        public Student(Student originalStudent) {
                for (Course c : originalStudent.getClasses()) {
                        classes.add(c);
                } // SHALLOW COPY: a reference to the Course is added, not a new separate object
        }         // If we modify the Course objects of the originalStudent, our new Student's
 }                // Course objects would also change
```

DEEP COPY

```
public class Student {
        private ArrayList<Course> classes = new ArrayList<>();

        public Student(Student originalStudent) {
                for (Course c : originalStudent.getClasses()) {
                        classes.add(new Course(c));
                } // DEEP COPY: a new object is created and added to classes
        }         // If we modify the Course objects of the originalStudent, our new Student's
 }                // Course objects would NOT change
```