COP 3330, Spring 2013

Exam 2 Review

Instructor : Arup Ghosh
03-25-13

School of Electrical Engineering and Computer Science
University of Central Florida

Exam 2

- Friday, March 29.
- In class, 50 minutes long.
- No aids of any kind.
- Things to bring
  - Yourself
  - PEN
  - ID

Things *not* on the exam

- GUIs

Exam 2 format

- Two sections – Total : 100 points
- First Section – Multiple Choice / TF (20*2 = 40)
- (Quiz 1 to 7?)

- Second Section – Free response (60)

Horribly lame geek joke

```
class Exam2Review extends Exam1Review {
      public Exam2Review() {
            super();    // i.e., Exam 1 Review
            stuffSinceThen();
      }
}
```

- So basically, go read the review for the previous exam again – today I'll just review things we've done since that time.

Essential topics

- Interfaces, inheritance, abstract classes
- Using the Comparable interface to sort objects
- The modifiers final and protected
- Polymorphism (is-a)
- Collections
- Exceptions, errors
- Enums

Polymorphism

- Polymorphism revolves around the is-a relationship of data types.
  - Recall that classes are types
  - Java polymorphism only really applies to classes, since only they have inheritance/interfaces
  - Primitives aren't polymorphic

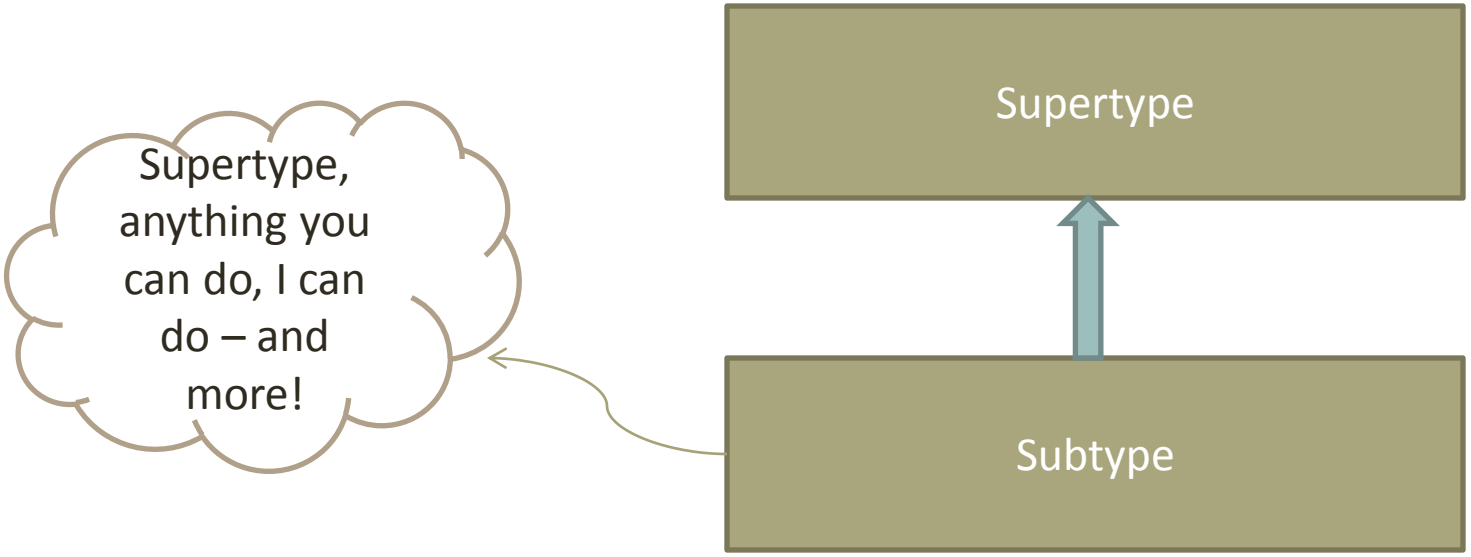
- Java allows the creation of is-a relationships in two ways
  - Interfaces
  - Inheritance

Liskov substitution

- The **Liskov Substitution Principle** is a nice way of summarizing the essence of polymorphism.
- "If S is-a T, then you can use an S in any place you can use a T."
- Thus you can *substitute* an object of a subtype for an object of a supertype.
  - Not the other way around!
- The subtype S also has the type T. This 'other form' is the reason we say it is **polymorphic** (*multiple forms*).

Substitution

- So for inheritance…



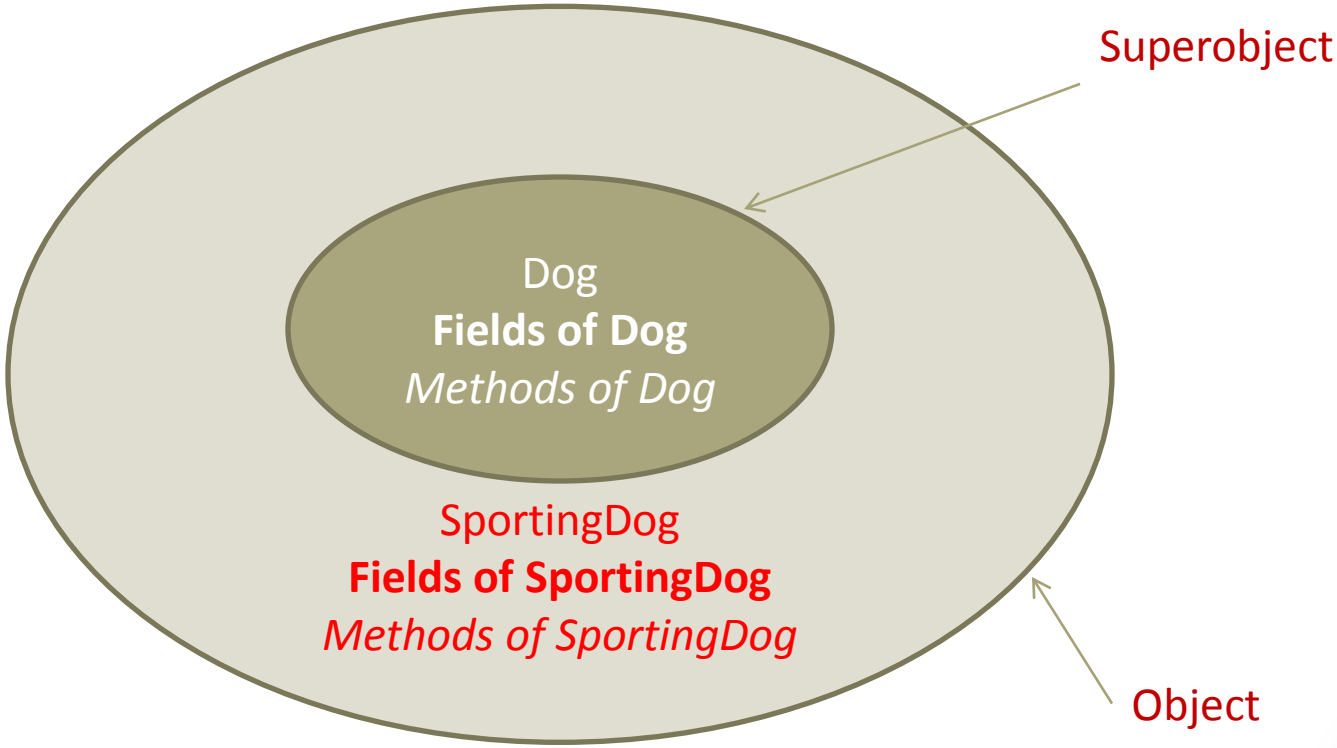
- This is basically the superclass/subclass relation. The subclass *inherits* the abilities of the subtype, but can also do other things, or do the inherited things differently (overriding).

Inheritance

- A subclass S **extends** its superclass T.
- This means it **inherits** all the non-private members of T.
  - i.e., it can freely access those members as its own
- Structurally, every S object is constructed around a core T object (the *superobject*, if you will).
  - This is why we start constructors by calling the superconstructor – we need to initialize the foundational superobject before building anything on top of it.
- Because S is everything T is – plus some more stuff added on.

Structure

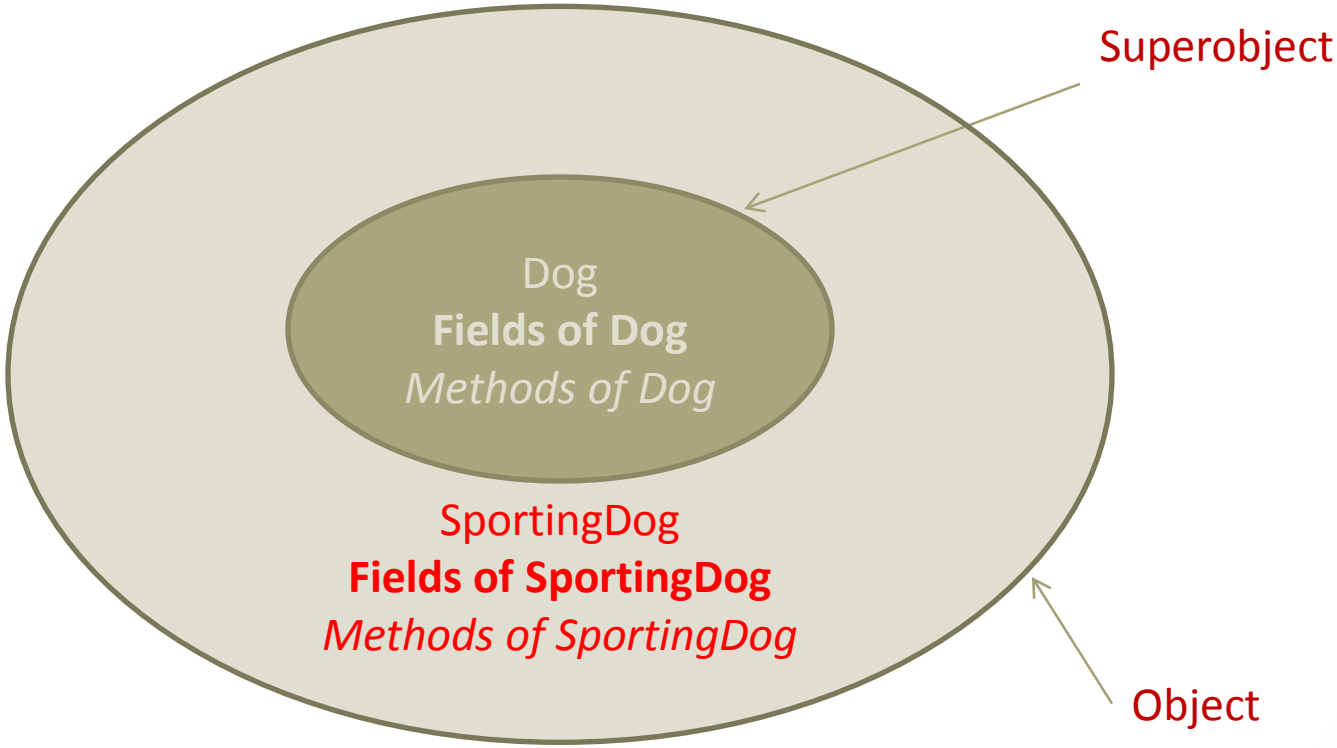
The superobject is the thing being referenced by the super keyword.



Just as the object is the thing being referenced by the this keyword.

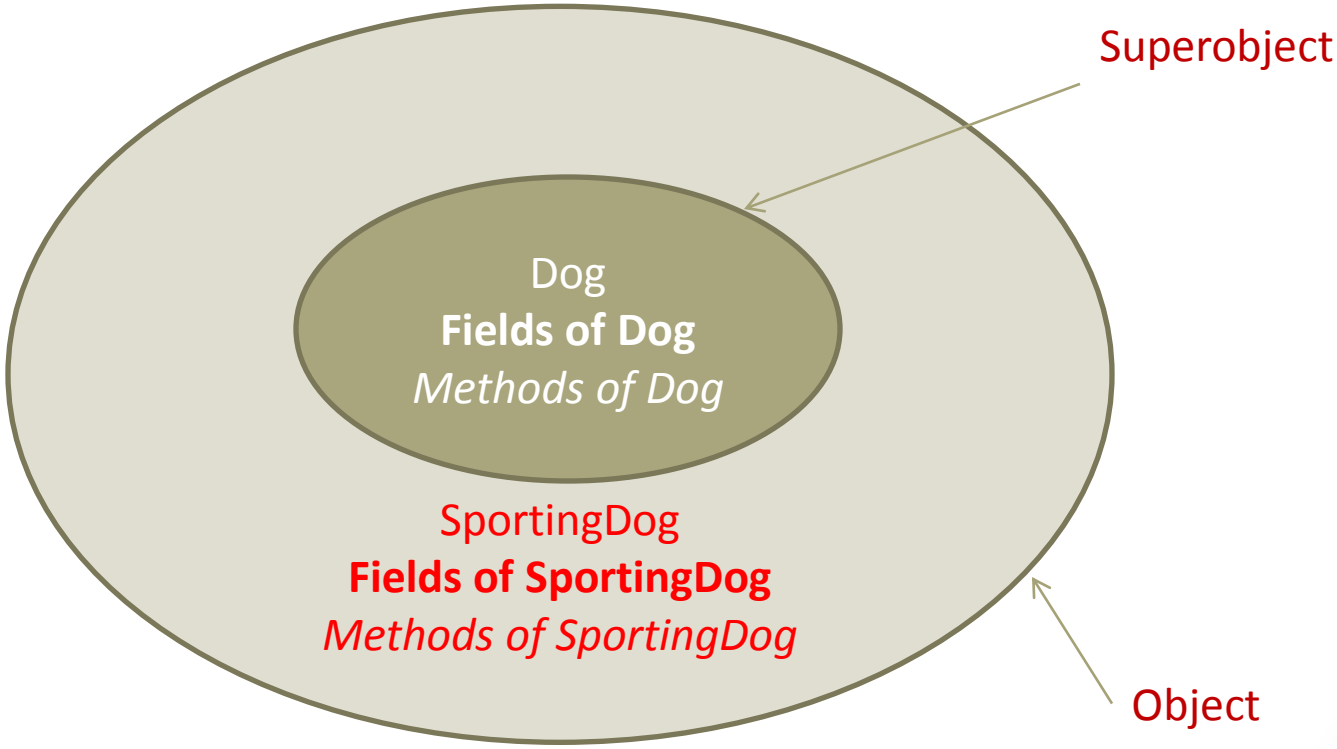
Structure

The superobject has to be initialized first, hence the requirement for a superconstructor call in any constructor.



Structure

The public members of the superobject ‘shine through’ to the outside, along with the public members of the object.



Interfaces

- Inheritance is a very substantial form of is-a relationship. The superclass actually gives something to the subclass.
- In contrast, interfaces are all about *appearances*.
- An interface defines a series of abstract methods (no bodies, just names, return types and parameters)
- To make a class implement an interface, use the implements keyword and provide bodies for each of these methods.

Interface polymorphism

- If class S implements interface T, then S is-a T.
- But note that it's because every S *looks like* a T.
  - T is defined by those methods (it's external appearance).
  - When it implements T, S is forced to add those methods to itself.
- For instance, there have been 21 Phantoms since 1536. But to outsiders, they're all the Phantom, because they 'look the same' to everyone.
- Local tribesmen (client objects) think it's the same man.

Abstract classes

- A hybrid of interfaces and normal classes.
- Basically, classes with abstract methods.
- Like interfaces, you cannot instantiate them. They have missing method bodies, after all. Only complete classes can be instantiated.
- You can inherit from them normally, but unless the subclass is also abstract, you must provide bodies for all abstract methods.
- Basically, abstract classes allow us to be *intentionally vague*, leaving things out so they can be filled in by some subtype closer to the problem.

Substitution again

- How does the substitution rule work? It boils down to this kind of assignment statement:
  - `superTypeVar = subTypeObj;`
- For instance, an Animal variable can contain a Dog object.
- Why? Because as a subclass, a Dog can do anything an Animal can.
- So I can call whatever methods I like on `superTypeObj`, and it will never fail because `subTypeObj` has the same methods.
  - It can pretend to be its supertype perfectly.

Substitution again

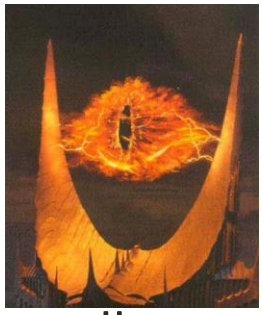
- Note that I said *type*, not class. The same rule applies for interfaces, which are not classes, but *are* types.

- However, interfaces cannot be instantiated (ditto for abstract classes). If I have an interface `Noisy`, I can't say `Noisy x = new Noisy();`
  - There's nothing to instantiate! Interfaces contain no data, and their methods are abstract. Direct instantiation is meaningless.

- Still, we can make variables that have the **type** `Noisy`. But only objects of classes that implement `Noisy` can be stored in them. (Same goes for abstract classes)
  - Why? Because they can perfectly pretend to be `Noisy`.

Taking it further

- That single assignment results in some pretty rich behavior.
- E.g., a method that takes a parameter of type Animal will work perfectly with Dog, Cat, Cow, etc.
  - “One method to rule them all…”



- When in doubt, just remember that this is what all polymorphic behavior boils down to.

The protected modifier

- Private members aren't inherited the same way as others.
  - They're still there, in the superobject, but the object has no way to access them.

- There's a tradeoff – sometimes you want to make members private (for information hiding) but you also want them to be inherited by subclasses.

- The solution is the `protected` modifier. A protected member is essentially public to subclasses, but private to anyone else.

The final keyword

- Final is used to make constants when applied to variables.
- When applied to method declarations, it means that a subclass **cannot** override that method.
- When applied to class declarations, it means that it is impossible to extend that class. No subclasses.

Collections

- Read the examples to learn the basics of Lists, Maps and Sets.
- Maybe skim through the documentation or additional examples on the Java website.
- Make sure you understand what they're for, since you'll probably have to use them in some way on the exam.

Exceptions

- Know how to use `try-catch, throws and throw`
- Remember that descendants of `RuntimeException` are *unchecked.*
- If your code contains a method call that might throw an exception (indicated by throws), the compiler will **force** you to handle it with `try-catch` or with `throws`.
- However, this doesn't apply for unchecked exceptions. They can still crash your program, but the compiler will not enforce exception handling.

Types of errors

- Compile-time
- Run-time
- Logical

- Make sure you understand the differences and can identify which is which.
  - E.g., a syntax error is a compile-time error
  - A NullPointerException is a runtime error.
  - When your chess AI starts to manufacture cyborg killing machines and sending them back in time, that's probably a logical error.

Enums

- We've used these pretty recently, so they should still be somewhat fresh.
- Enums are basically type-safe constants.
- We use them whenever we want to turn something simple, with only a small number of possible values, into a type.
- The alternative is representing them with something like constant ints, which is error-prone.

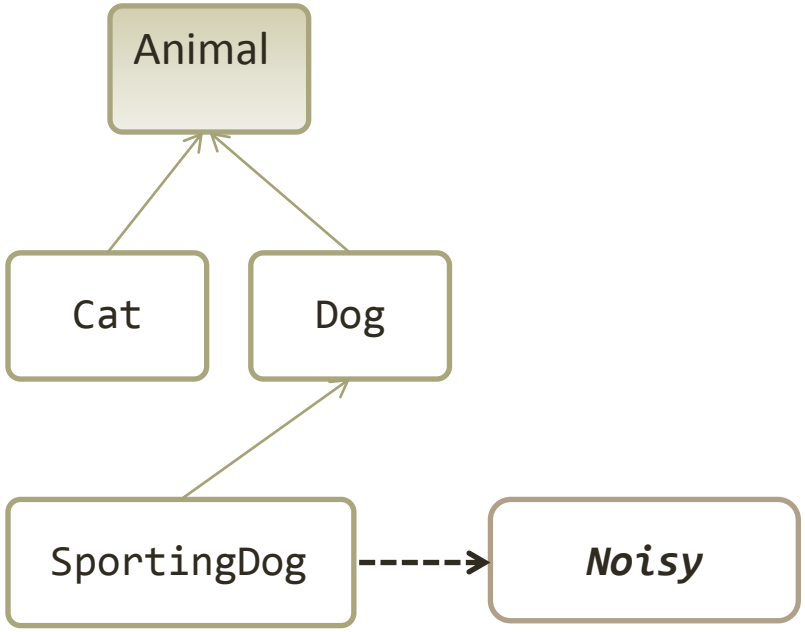
Useful contrasts

- Overriding vs overloading
- Abstract classes vs interfaces
- Static members vs instance members
- super vs this
- protected vs private

- Comparing things in this way is often useful for solidifying understanding. ☺

Sample Questions

- Questions on inheritance hierarchy.



Sample Questions

- Fill in the blank methods for the class below, according to the instructions in the comments.

```
class Dog implements Comparable<Dog> {
        //variables
        private String name;
        private double height;
        public Dog (String name, double height) {
           this.name = name;
          this.height = height;
        }
        //Write getter method for Name
        public String getName() {

        }
        ……..
        //instructions for compareTo()
        public int compareTo (Dog that) {

        }
}
```

Sample Questions

- Questions from Collections
- ArrayList
  - Find average height
- TreeSet
  - Unique names
- TreeMap

Sample Questions

- Exception classes
- try catch block

```
Scanner scanner = new Scanner(System.in);
try {
        System.out.println("Enter a integer: ");
         int number = scanner.nextInt();
        System.out.println("The number you entered is:" + number);
}
catch (InputMismatchException ex) {
        System.out.println("Try again, incorrect input: Integer required.");
        scanner.nextLine();
        throw ex;
}
catch(NoSuchElementException ex) {
       System.out.println("Improper format!");
}
```

.....

Sample Questions

- Given an abstract class:

```
public abstract class Pet {
  public abstract void makeNoise();
}
```

- What methods would the following class need?

```
public class Cat extends Pet
     implements Comparable<Cat>
```

Sample Questions

- Write a method that takes in an int array and a boolean array of the same size, and returns an array consisting only of values from the first array such that the corresponding element in the boolean array is true
  - `[1,2,7,8,9,4]`
  - `[t,t,f,t,f,t]`
  - `=> [1,2,8,4]`

Sample Questions

Suppose that a program is frequently crashing on the line:

```
c = b / a;
```

with the following message:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at MyProgram.myMethod(MyProgram.java:166)
        at MyProgram.main(MyProgram.java:29)
```

Suggest two different simple modifications that could be made to the program to prevent it from crashing while still having that line be reachable.

Sample Questions

- Write a public class Monkey that implements the Comparable<Monkey> interface.
  - (1) A Monkey contains two pieces of information: name (a String) and number of bananas eaten (an int).
  - (2) There should be two public constructors for this class. The first constructor takes in one parameter that initializes name of the Monkey, and sets number of bananas eaten to zero. The second constructor takes in two parameters, initializing both name and number of bananas eaten for the Monkey.
  - (3) After instantiation, a Monkey's name cannot be changed, and the number of bananas eaten can only increase.

- (4) There should be a public instance method eatBanana, which takes no parameters and increments the number of bananas eaten by one.
- (5) The natural ordering of Monkeys should rank Monkeys who ate more bananas before Monkeys who ate less. If two Monkeys have eaten the same number of bananas, then their order is determined by their names in lexicographical order. This means, in a sorted array of Monkeys, Monkeys who ate the most number of bananas comes first. Among Monkeys who ate the same number of bananas, their order is determined by their names using the compareTo method of Strings. The String comparison should be case sensitive.

Sample Questions

- UML diagrams
- Composition
- Aggregation
- Association

- Questions similar to Car – Engine Example