COMP2522

Lesson 3: final, abstract

Lesson 3 Topics

- Final Methods
- Final Classes
- Overriding .equals() method
- Abstract methods
- Abstract classes

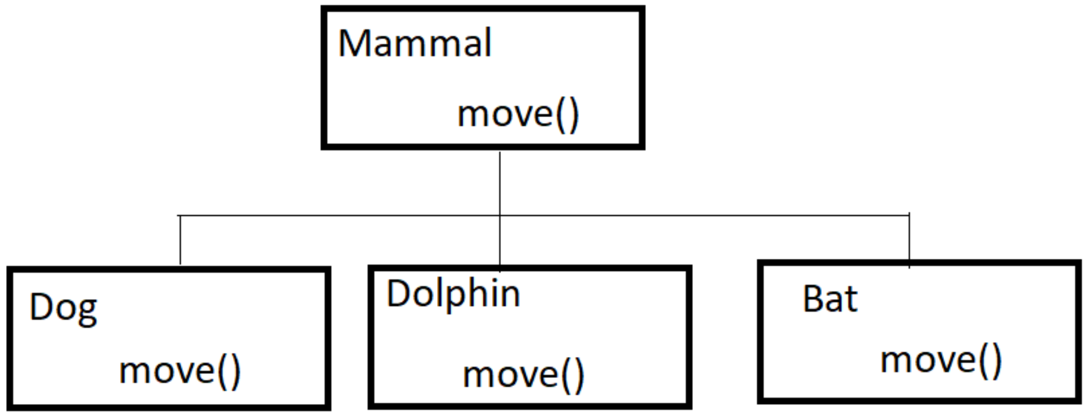
Review

- final Animal a;
- a = new Dog();

- at compile time, a is an Animal
- at run time, a is a Dog

- a's static type is "Animal"; its dynamic type is "Dog"

Runtime vs. Compile Time



```
Mammal m = new Dolphin();

// calls Dolphin-class move() method at runtime (dynamic checking)
// checks that Mammal-class move() method exists at compile time (static checking)
m.move();
```

m = new Bat(); // may happen

or if not, it does exist in
the parent (Object class)

Review

- Polymorphism is the dynamic (runtime) lookup of methods
- The three pillars of Polymorphism:

1. Inheritance	class Dog extends Mammal
2. Substitution	Mammal m = new Dog()
3. Overriding	m.speak() // "woof"

- **Static** type checks are performed by the compiler
- **Dynamic** type checks are performed by the JRE

"final" Methods

- Prevents subclasses from overriding that method
- Often, a parent class's setter methods are declared final, to ensure that <u>only that class</u> can mutate its own fields, on its own terms (its own rules)
- Subclasses should never be able to "directly" change a superclass field except via super() calls in the constructor
- Subclass setters should likewise be made final

"final" Classes

- Prevents subclasses from created
- A final class cannot be extended

equals() Method

- https://docs.oracle.com/en/java/javase/22/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)
- When comparing objects, Java (or a developer) can call its equals() method
- The Object class has an equals() method, which simply returns whether the two objects have the same address; i.e., is the parameter object actually *the same object* as "this"; in other words it acts the same as ==
- Note: do not use == with Strings (or other objects) unless you really do want to compare addresses
- It is often overridden so that we can determine what we mean by "these objects are equal":

```
@Override
public boolean equals(final Object o)
{
        if(o == null){
                return false;       // this is definitely not equal to null
        }
        if(!(o.getClass().equals(this.getClass()))){           // or instanceof
                return false;       // these aren't even the same class!
        }
        // now return true if and only if your criteria are satisfied
}
```

hashCode() method

- **Whenever you override equals(), you *must* also override hashCode()**
- **An object must return the same hashCode if called multiple times in running an application**
- The Object class has a hashCode() method, which simply returns a hash of the object's address
- It can be overridden:

```
@Override
public int hashCode(){
		// let the IDE override this for our sake
}
```

- **Important Rule: <u>equal objects</u> must return <u>equal hashCodes</u>**

Abstract methods

- An abstract method has no body at all
- "I won't tell you what to do, but my children <u>must"</u>
- public <u>abstract</u> void move(); // no curly braces {}

  or
- public <u>abstract</u> void move(int x, String y); // no curly braces {}

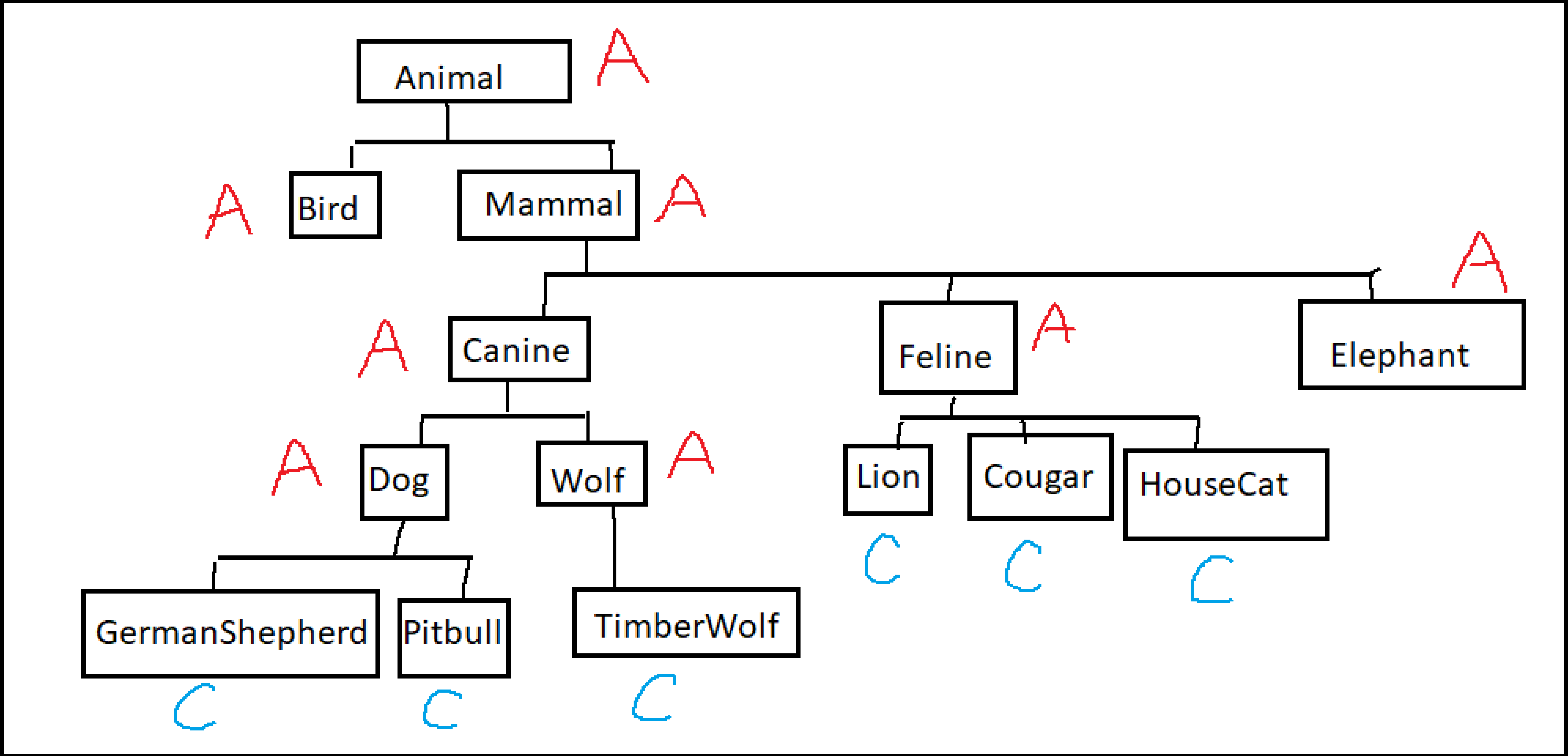
Abstract classes

1. If a class has one or more abstract methods, the class MUST be declared abstract

2. An abstract class cannot have objects made of it; it is just an *idea*

3. For a class to be concrete (i.e. not abstract; objects CAN be made of it), that class must implement (override) all parent (grandparent, etc…) methods which were abstract. In other words, if a parent has abstract methods, and its child doesn't override them, the child and the parent both must be abstract classes

Abstract: Just an Idea (not an object)



Abstract

- Any class that we know so far, at any time, can be made abstract
- EXCEPT final classes
- A final class means "no class can extend this one"
- Example: try extending the java.lang.String class (you can't)

- The opposite of an abstract class is a concrete class

- An abstract class can contain:

- only abstract methods, e.g. public abstract void foo()**;**
- only concrete methods, e.g. public void foo()**{}**
- a mix of abstract and concrete methods
- no methods
- instance variables, statics, constants
- constructors

Abstract

- If a parent class contains **public abstract void foo();**
- …that class must be abstract
- …the child class will also be abstract unless/until it implements foo(): public void foo()**{}**

- No object can be created of any class that has any abstract methods that have not been implemented (in that class or any ancestor)

Abstract

- We put abstract methods into a parent class so that Java has a <u>guarantee</u> that all subclasses definitely inherit and (eventually) implement that method
- Any time you have an object, you know that all abstract parent methods *must have been* implemented

- e.g Database class: abstract methods for open, close, connect, query….

Abstract

- Consider an abstract parent class A that contains an **abstract method foo()**
- Any child class of A:

- must be abstract too
- or
- must implement (override) foo()
- or
- both

- YOU CANNOT HAVE AN OBJECT THAT HAS AN UNDEFINED METHOD

Abstract

- Example:
- ChessPiece could be an abstract parent class
- Pawn would be a concrete child class

- Sport could be an abstract parent class
- Ice Hockey would be a concrete child class

- Weather could be an abstract class
- Rain would be a concrete child class

Abstract Class with Static Methods

- Another useful abstract class is for it to act as a container for static methods
- No object is required
- Let's make a static recursive method to perform multiplication

```
abstract class Math{
 public static int recMultiply(final int a,
                                              final int b) {
      if(b == 0) {
         return 0;
      }
      return a + recMultiply(a, b - 1);
   }
}
```