COMP241
Software Engineering Development
Lecture 7: Inheritance & Polymorphism

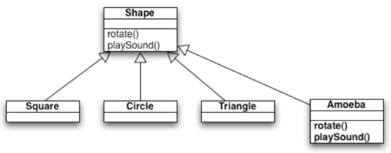
Mark Hall

Readings: Horstman Chap 6

- Understanding inheritance
- Avoiding code duplication
- Polymorphism
- Rules for overriding

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

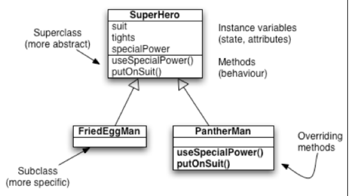
Specialization



- `Amoeba` class **overrides** the `rotate` and `playSound` methods of `Shape`
- Overriding
  - Subclass redefines one of the inherited methods when it needs to change or extend the behaviour of that method

Understanding Inheritance

- Subclass inherits from the superclass
  - Instance variables and methods
- In Java the subclass *extends* the superclass
- Instance variables are not overriden
  - Don't have to be as they don't define any special behaviour

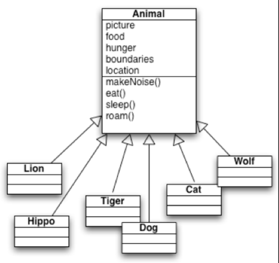


Animal Simulation Program

- Design the inheritance tree
  - Lion, Hippo, Tiger, Dog, Cat, Wolf
- Abstract out behaviours
  - What do these six types have in common?
- Define inheritance tree relationships
  - How are these types related?

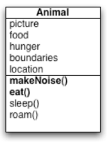
Using inheritance to avoid duplicating code in subclasses

- Design a class that represents common state and behaviour
- Put in methods an instance variables that all animals might need



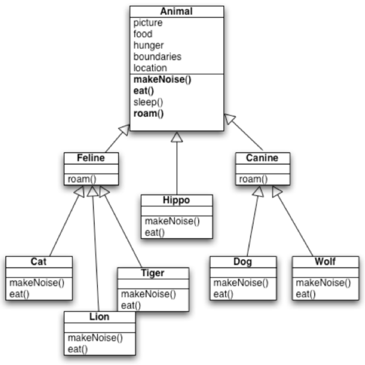
Inheritance opportunities

- Decide if a subclass needs behaviours that are specific to that subclass type
  - Looking at the `Animal` class, we decide that `makeNoise` and eat should be overriden in subclasses
- Further opportunities for abstraction
  - `Wolf` and `Dog` might have behaviour in common, same for `Tiger`, `Lion` and `Cat`

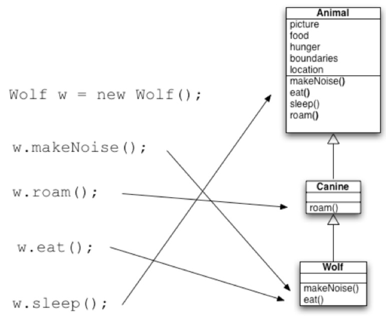


Inheritance opportunities

- Finish the class hierarchy
- Decide that Canines could use a common roam method
  - Move in packs
- Same for Felines
  - Avoid others of their kind
- Hippo continues to use generic roam



Which method is called?



```
Wolf w = new Wolf();

w.makeNoise();

w.roam();

w.eat();

w.sleep();
```

Using IS-A and HAS-A

- When you want to know if one thing should extend another, apply the IS-A test
  - Triangle IS_A Shape, Cat IS-A Feline, Surgeon IS-A Doctor
- Bathroom HAS-A Tub
  - Bathroom has a Tub instance variable
- If class B extends class A, then class B IS-A class A
  - True anywhere in the inheritance tree
  - If class C extends class B, then C passes the IS-A test for both B and A

Inheritance dos and don'ts

- Do use inheritance when one class is a more specific type of a superclass
- Do use inheritance when you have behaviour (implemented code) that should be shared among multiple classes of the same general type
- Don't use inheritance just so that you can reuse code, if the relationship between the subclass and the superclass violate either of the above two rules
  - E.g. Alarm class has printing code, need printing in Piano class, so extend Alarm
- Do not use inheritance if the subclass and superclass do not pass the IS-A test

Advantages

- You avoid duplicate code
  - When you wan't to change behaviour, you only have to modify it in one place
- You define a common protocol for a group of classes
  - All subclasses under a certain supertype have all the methods that the supertype has
    - **This is cool** because you get to take advantage of **polymorphism**
    - **And this matters** because you get to refer to a subclass object using a reference declared as the supertype
    - **And that means** you get to write flexible code—easier to develop and easier to extend

Polymorphism

- Lets look at how we normally declare a reference and create an object
  1. Declare a reference variable
     `Dog myDog = new Dog();` (**Dog myDog** in bold)
     – Tells the VM to allocate space for a reference variable of type `Dog`
  2. Create an object
     `Dog myDog = new Dog();` (**new Dog();** in bold)
     – Tells the VM to allocate space for a new `Dog` object on the heap
  3. Link the reference and the object
     `Dog myDog = new Dog();` (**=** in bold)
- The important part is that the reference type and the object type are the same—`Dog`

Polymorphism

- But with polymorphism, the reference and the object can be **different**
  - **`Animal`** `myDog = new` **`Dog();`**
  - The reference type can be a supertype of the actual object type

Polymorphism

```
Animal[ ]  animals = new Animal[ 5] ;


animals[ 0]  = new Dog();
animals[ 1]  = new Cat();
animals[ 2]  = new Wolf();
animals[ 3]  = new Hippo();
animals[ 4]  = new Lion();


for (i = 0; i < animals.length; i++) {
  animals[ i] .eat();
  animals[ i] .roam();
}
```

Declare an array of type Animal

You can put any subclass of Animal in the array!

And here is the best polymorphic part - you get to loop through the array and call one of the Animal-class methods, and every object does the right thing!!

Same with roam().

When 'i' is 0 (a Dog is at index 0), you get Dog's eat method. When 'i' is 1, you get Cat's eat() method

Polymorphism

- Of course, you can also have polymorphic arguments and return types

```
class PetOwner {
  public void start() {
    Vet v = new Vet();
    Dog d = new Dog();
    Hippo h = new Hippo();
    v.giveShot(d);
    v.giveShot(h);
  }
}
class Vet {
  public void giveShot(Animal a) {
    // do horrible things to the animal at
    // the other end of the 'a' parameter
    a.makeNoise();
  }
}
```

The Animal parameter can take ANY Animal type as the argument. When the Vet is done giving the shot, it tells the Animal to makeNoise(), and whatever Animal is really out there on the heap, that's whose makeNoise() method will run.

Polymorphism

- With polymorphism, you can write code for methods that doesn't have to change when you introduce new subclass types into the program
  - You can write code, go on vacation, and someone else can add new subclass types to the program and your methods will still work.

Bits and Pieces

- A subclass that overrides a superclass method can still access the superclass implementation

```
public void roam() {
  super.roam(); // call the inherited version
  // my own roam stuff
}
```

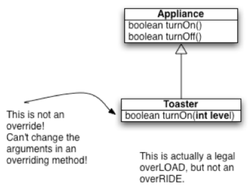
  - Superclass methods can be designed so that they will work for any subclass, even though the subclass may still need to 'append' more code
- A subclass can inherit `public` (and `protected`) members from its superclass but not `private` members

Bits and Pieces

- Non-public classes can only be subclassed by classes in the same package as the class
- Using the `final` modifier prevents a class from being subclassed
- Individual methods can be marked `final` (rather than the whole class) to prevent them from being overridden

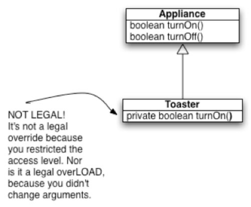
Keeping the contract: Rules for overriding

- Arguments and return types must be the same


Appliance
boolean turnOn()
boolean turnOff()
Toaster
boolean turnOn(int level)
This is not an override! Can't change the arguments in an overriding method!
This is actually a legal overLOAD, but not an overRIDE.

Keeping the contract: Rules for overriding

- The method can't be less accessible


Appliance
boolean turnOn()
boolean turnOff()
Toaster
private boolean turnOn()
NOT LEGAL! It's not a legal override because you restricted the access level. Nor is it a legal overLOAD, because you didn't change arguments.

Overloading a method

- Two methods with the same name but different argument lists
- An overloading method isn't trying to fulfill the polymorphism contract defined by its superclass
  - The return types can be different (as long as the argument lists are different)
  - You can't change just the return type
  - You can vary the access levels in any direction