COMP241
Software Engineering Development
Lecture 8: More on Polymorphism

Mark Hall

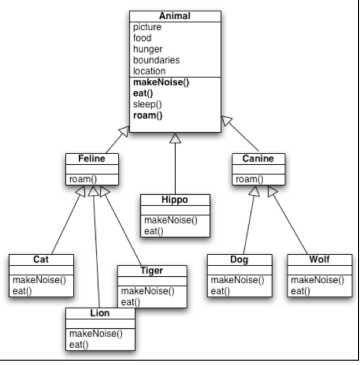
Readings: Horstman Chap 4 & 6

- Abstract classes
- Abstract vs. Concrete
- Abstract methods
- Changing the contract
- Multiple inheritance
  - Deadly Diamond of Death
- Interfaces

THE UNIVERSITY OF WAIKATO
DEPARTMENT OF COMPUTER SCIENCE
TARI ROROHIKO

Returning to the Animal hierarchy

- Have we forgotten something?
  - Class structure is not too bad
  - Duplicate code is kept to a minimum
  - Overridden the methods that should have subclass-specific implementations



Returning to the Animal hierarchy

- We know we can say:

```
Wolf aWolf = new Wolf();
Animal aHippo = new Hippo();
```

- Here is where it gets a bit weird:

```
Animal anim = new Animal();
```

- What does a new `Animal()` object look like?

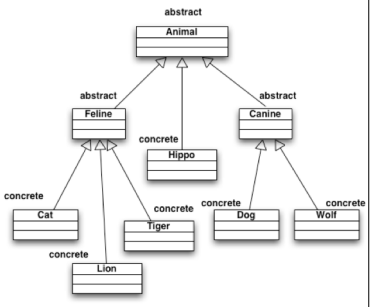
Abstract classes

- Some classes should never be instantiated
- By marking a class as `abstract` the compiler will stop code from creating an instance of that type
  - An abstract class has virtually no use, no purpose unless it is *extended*
  - With an abstract class, the guys doing the work at runtime are instances of a **subclass** of your abstract class

```
abstract class Canine extends Animal {
  public void roam() { }
}
```


Abstract vs. Concrete

- A class that is not abstract is called a concrete class



Abstract methods

- An abstract class means that the class must be extended
- An abstract method means that the method must be overridden
  - Exists solely for *polymorphism*
- If you declare an abstract method, then you must mark the class abstract as well
  - Can mix both abstract and non-abstract methods in an abstract class

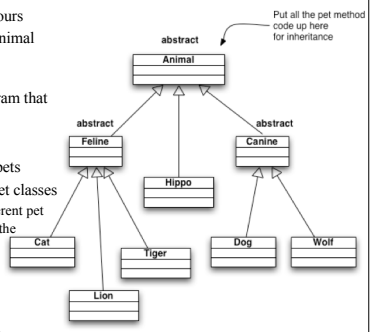
Changing the contract

- We've seen how much Java cares about the methods in the class of the *reference* variable
  - You can call a method in an object only if the class of the reference variable has that method
  - The compiler checks the *reference* variable, not the actual *object* at the other end of the reference

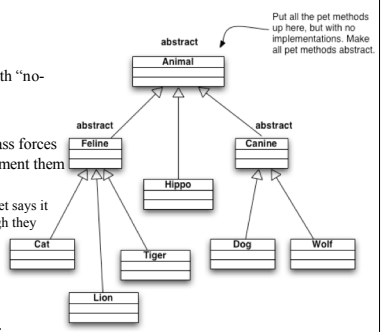
Changing the contract

- Say we have a `Dog` object…
  - The `Dog` class isn't the only contract that defines Dogs—public methods are inherited from all of Dog's superclasses
- If `Dog` was defined with the Animal simulation in mind then we could probably reuse `Dog` for a Science Fair Tutorial on `Animal` objects
- But what if we wanted to use `Dog` for a PetShop program?
  - We don't have any `Pet` behaviours: `beFriendly()`, `play()`

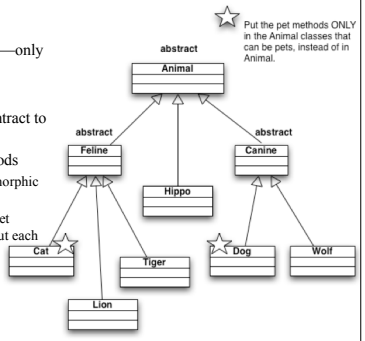
- Option 1:
  - Put all the pet methods in `Animal`
- Pros:
  - All animals inherit pet behaviours
  - Don't have to touch existing animal subclasses
  - `Animal` can be used as a polymorphic type in any program that wants to use pets
- Cons:
  - Some animals are clearly not pets
  - Will still have to modify the pet classes
    - E.g. `Dog` and `Cat` have different pet behaviours; have to override the defaults



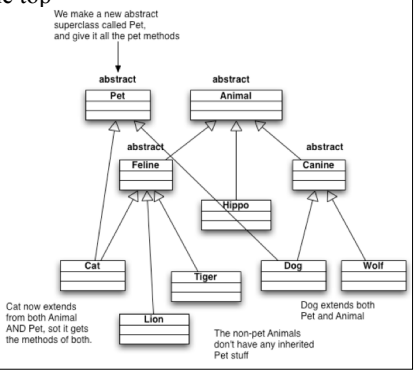
- Option 2:
  - Start with Option 1, but make the Pet methods abstract
- Pros:
  - Benefits of Option 1, but all subclasses **must** override
  - So, non-pets can override with "no-op" methods
- Cons:
  - Abstract methods in superclass forces concrete subclasses to implement them
    - Time consuming
    - Bad contract—every non-pet says it has pet methods even though they don't do anything!



- Option 3:
  - Put the pet methods ONLY in the classes where they belong
- Pros:
  - Methods are where they belong—only in pet classes
- Cons:
  - Protocol is not explicit—no contract to back it up
  - No polymorphism for pet methods
    - Can't use Animal as the polymorphic type
    - Every class that wants to use pet behaviours needs to know about each and every pet class

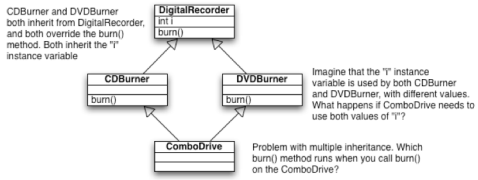


- It looks like we need two superclasses at the top



Multiple Inheritance

- Multiple inheritance can be a really bad thing
  - If it was possible in Java that is
- Deadly Diamond of Death!!

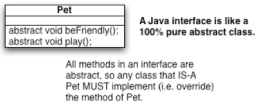


Interfaces

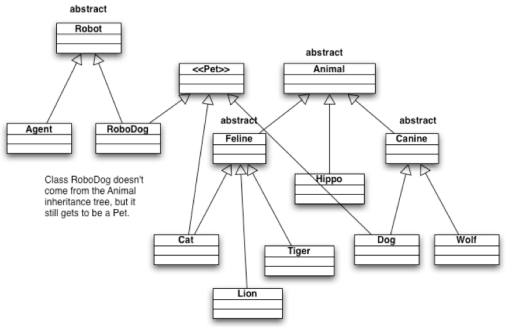
- Interfaces provide the solution
  - Not GUI interfaces
  - Not generic interfaces as in the public interface for class X
  - Java *keyword* `interface`
- Provides much of the polymorphic benefits of multiple inheritance without the pain and suffering of DDD (Deadly Diamond of Death)

Interfaces

- Interfaces side step DDD by making all methods **abstract**
  - Concrete subclass must implement the methods
  - At runtime the JVM is not confused about which of the two inherited versions it is supposed to call



- Classes from different inheritance trees can implement the same interface
  - Better still, a class can implement multiple interfaces

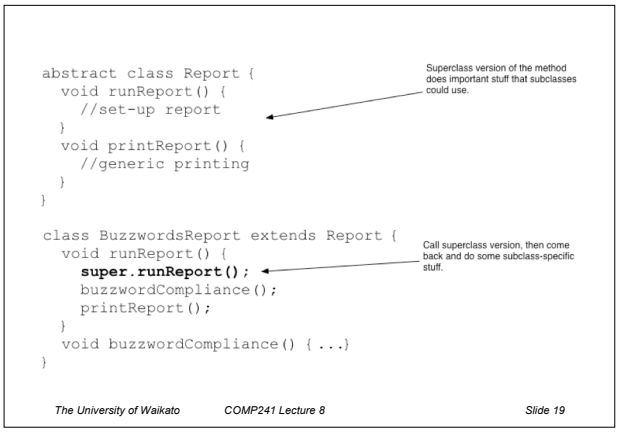


Class, subclass, abstract class or interface?

- Make a class that doesn't extend anything (except for Object) when your new class doesn't pass the IS-A test for anything
- Make a subclass only when you need a **more specific** version of a class and need to override or add new behaviours
- Use an abstract class when you want to define a **template** for a group of classes
  - You have at least some implementation code that all subclasses could use
  - Make the class abstract when you want to guarantee that nobody can make objects of that type
- Use an interface when you want to define a **role** that other classes can play regardless of where those classes are in the inheritance tree

Invoking the superclass version of a method

- A concrete subclass that wants to **add** to a superclass method, not replace completely


abstract class Report {
void runReport() {
//set-up report
}
void printReport() {
//generic printing
}
}
Superclass version of the method does important stuff that subclasses could use.
class BuzzwordsReport extends Report {
void runReport() {
super.runReport();
buzzwordCompliance();
printReport();
}
void buzzwordCompliance() { ...}
}
Call superclass version, then come back and do some subclass-specific stuff.
The University of Waikato
COMP241 Lecture 8
Slide 19