Inf1B

Inheritance B

Volker Seeker
adapting earlier version by Perdita Stevens and Ewan Klein

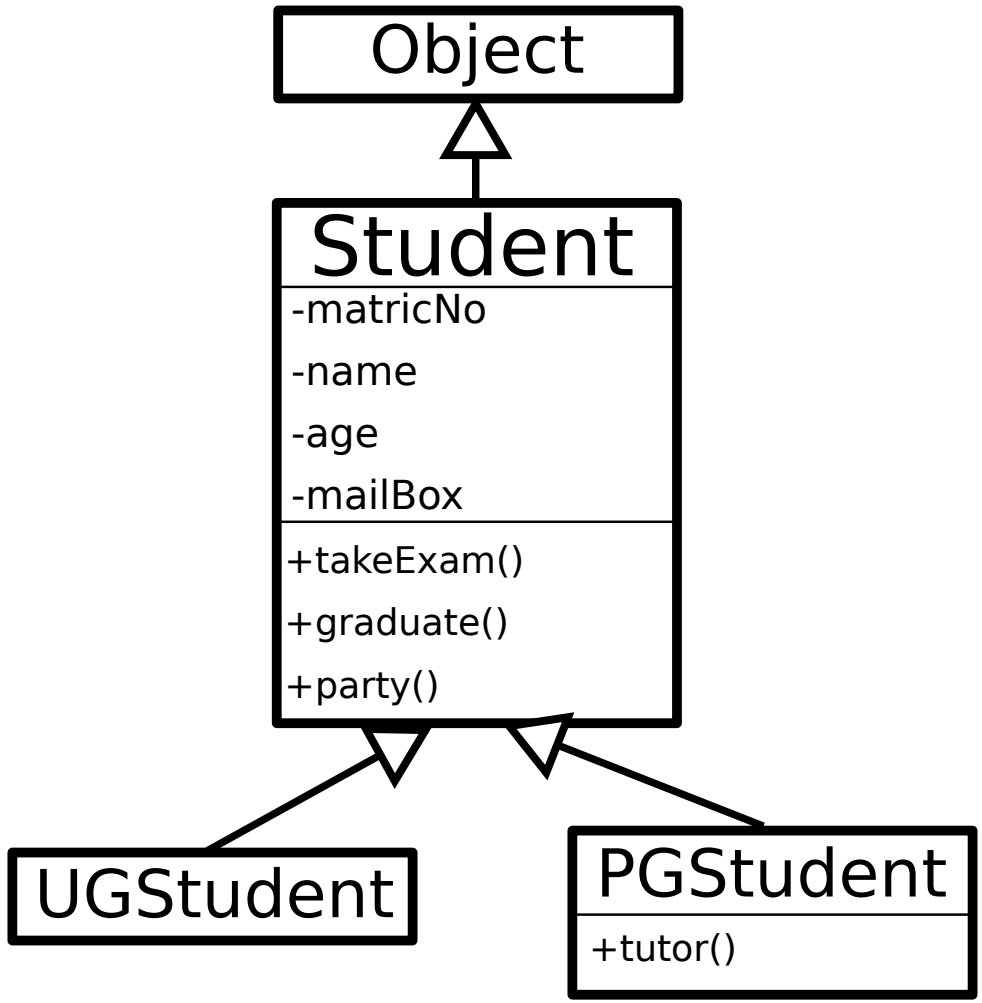
School of Informatics

February 19, 2020

Abstracting Common Stuff

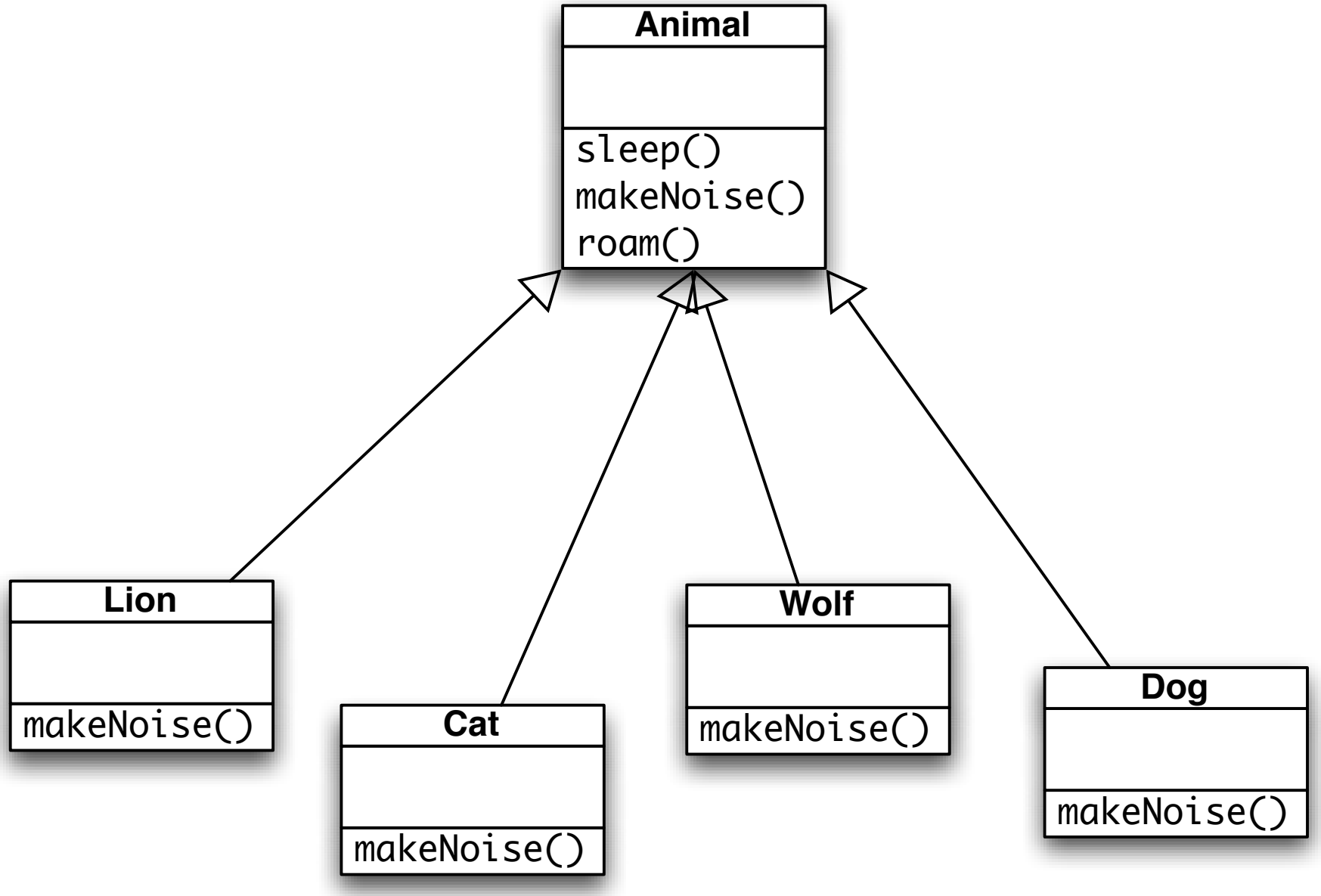
Inheritance hierarchy:

Subclass (`UG`, `PG`) inherit from superclass (`Student`) inherits from superclass (`Object`)



Flat vs. Nested Hierarchies

Flat Animal Hierarchy



Animals Example, 1

Our base class: `Animal`

Animal

```
public class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public void makeNoise() {
        System.out.println("Noises...");
    }
    public void roam() {
        System.out.println("Roamin' on the plain.");
    }
}
```

Animals Example, 2

1. `Lion subclass-of Animal`
2. Override the `makeNoise()` method.

Lion

```
public class Lion extends Animal {
    public void makeNoise() {
        System.out.println("Roaring: Rrrrrr!");
    }
}
```

Animals Example, 3

1. `Cat subclass-of Animal`
2. Override the `makeNoise()` method.

Cat

```
public class Cat extends Animal {
    public void makeNoise() {
        System.out.println("Miaowing: Miaooo!");
    }
}
```

Animals Example, 4

1. `Wolf subclass-of Animal`
2. Override the `makeNoise()` method.

Wolf

```
public class Wolf extends Animal {
    public void makeNoise() {
        System.out.println("Howling: Ouoooooo!");
    }
}
```

Animals Example, 5

1. `Dog subclass-of Animal`
2. Override the `makeNoise()` method.

Dog

```
public class Dog extends Animal {
    public void makeNoise() {
        System.out.println("Barking: Woof Woof!");
    }
}
```

Animals Example, 6

The Launcher

```
public class AnimalLauncher {
    public static void main(String[] args) {
        System.out.println("\nWolf\n=====");
        Wolf wolfie = new Wolf();
        wolfie.makeNoise() ; // from Wolf
        wolfie.roam(); // from Animal
        wolfie.sleep(); // from Animal

        System.out.println("\nLion\n=====");
        Lion leo = new Lion();
        leo.makeNoise(); // from Lion
        leo.roam(); // from Animal
        leo.sleep(); // from Animal
    }
}
```

Animals Example, 7

Output

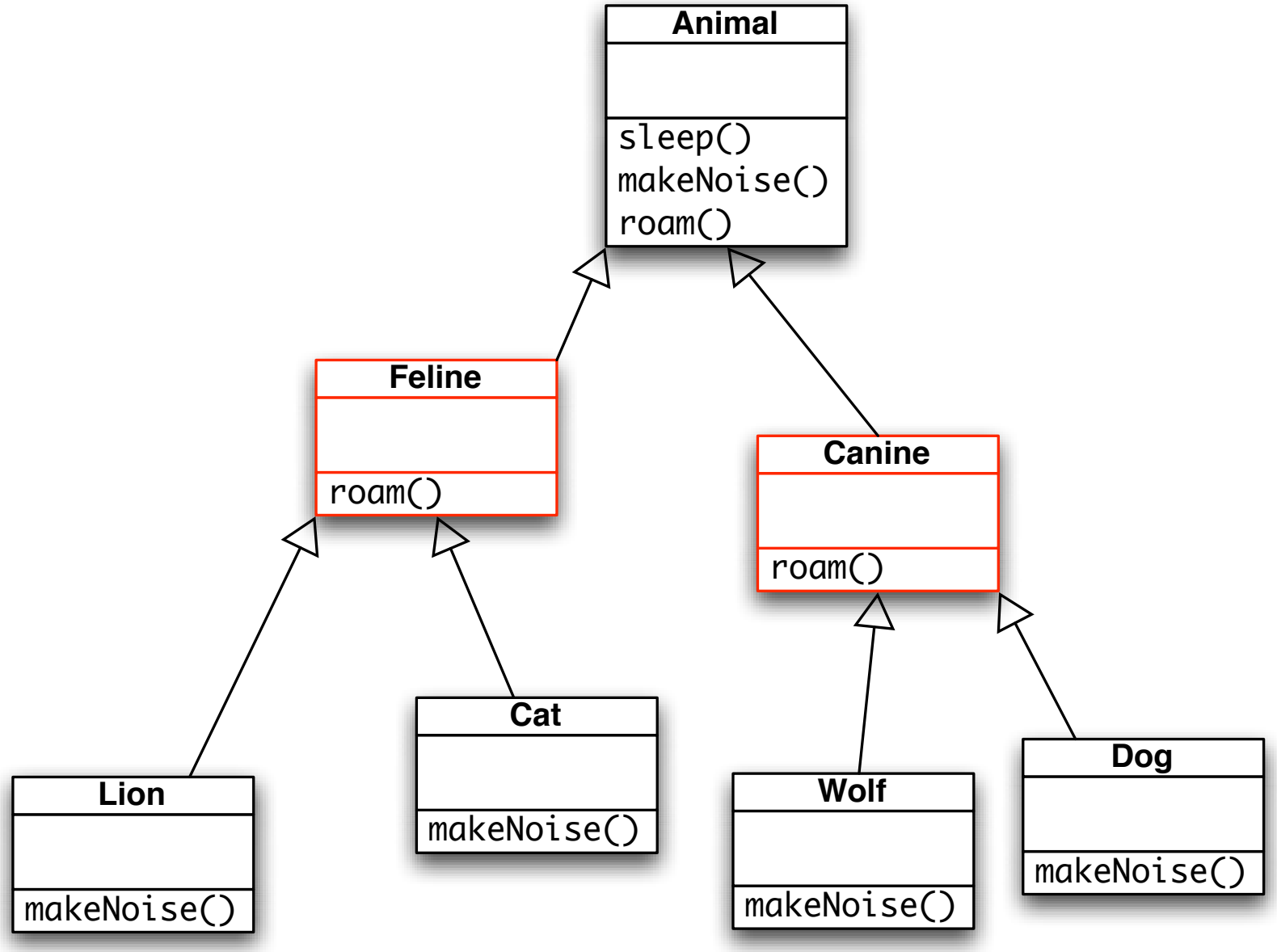
```
Wolf
=====
Howling: Ouoooooo!
Roamin' on the plain.
Sleeping: Zzzzz

Lion
=====
Roaring: Rrrrrr!
Roamin' on the plain.
Sleeping: Zzzzz
```

Nested Animal Hierarchy

- Lions and cats can be grouped together into `Felines`, with common `roam()` behaviours.
- Dogs and wolves can be grouped together into `Canines`, with common `roam()` behaviours.

Nested Animal Hierarchy



Animals Example, 1

Same as before.

Animal

```
public class Animal {
    public void sleep() {
        System.out.println("Sleeping: Zzzzz");
    }
    public void makeNoise() {
        System.out.println("Noises...");
    }
    public void roam() {
        System.out.println("Roamin' on the plain.");
    }
}
```

Animals Example, 2

The new class `Feline`

Feline

```
public class Feline extends Animal {
    public void roam() {
        // Override roam()
        System.out.println("Roaming: I'm roaming alone.");
    }
}
```

Animals Example, 3

The new class `Canine`

Canine

```
public class Canine extends Animal {
    public void roam() {
        // Override roam()
        System.out.println("Roaming: I'm with my pack.");
    }
}
```

Animals Example, 4

1. `Lion subclass-of Feline`
2. Override the `makeNoise()` method.

Lion

```
public class Lion extends Feline {
    public void makeNoise() {
        System.out.println("Roaring: Rrrrrr!");
    }
}
```

- Similarly for `Cat`.

Animals Example, 5

1. `Wolf subclass-of Canine`
2. Override the `makeNoise()` method.

Wolf

```
public class Wolf extends Canine {
    public void makeNoise() {
        System.out.println("Howling: Ouoooooo!");
    }
}
```

- Similarly for `Dog`.

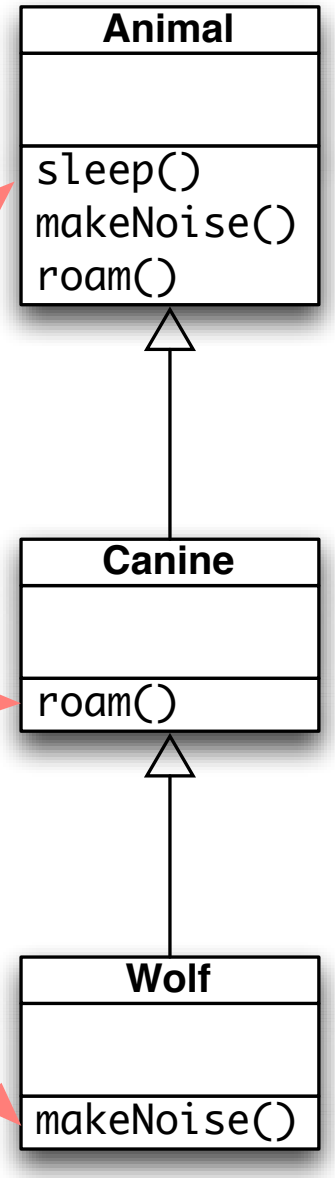
Which method gets called?

```
1. Wolf wolfie = new Wolf();

2. wolfie.makeNoise();

3. wolfie.roam();

4. wolfie.sleep();
```



Animals Example, 6

The Launcher

```
public class AnimalLauncher {
    public static void main(String[] args) {
        System.out.println("\nWolf\n=====");
        Wolf wolfie = new Wolf();
        wolfie.makeNoise(); // from Wolf
        wolfie.roam(); // from Canine
        wolfie.sleep(); // from Animal

        System.out.println("\nLion\n=====");
        Lion leo = new Lion();
        leo.makeNoise(); // from Lion
        leo.roam(); // from Feline
        leo.sleep(); // from Animal
    }
}
```

Animals Example, 7

Output

```
Wolf
=====
Howling: Ouoooooo!
Roaming: I'm with my pack.
Sleeping: Zzzzz

Lion
=====
Roaring: Rrrrrr!
Roaming: I'm roaming alone.
Sleeping: Zzzzz
```

Polymorphism

The ability of an object to take on many forms.

Declaring, Initialising and Using a Reference Variable

```
private static void goToBed(Animal tiredAnimal) {
    tiredAnimal.sleep();
}

public static void main(String[] args) {
    Animal myAnimal = new Animal();
    goToBed(myAnimal);
}
```

I am working only with the superclass here.

Declaring, Initialising and Using a Reference Variable

Polymorphism means:

I can use the subtype `Wolf` of the object `Animal` in any context where an object of type `Animal` is expected.

Declaring, Initialising and Using a Reference Variable

```
private static void goToBed(Animal tiredAnimal) {
    tiredAnimal.sleep();
}

public static void main(String[] args) {
    Animal myAnimal = new Wolf();
    goToBed(myAnimal);
}
```

The subclass can do **at least** everything the superclass can do.

(maybe a bit different though)

Formal Notation: `Wolf <: Animal` (Wolf is a subtype of Animal)

Polymorphic ArrayList

The Launcher

```
public class AnimalLauncher2 {
    public static void main(String[] args) {
        Wolf wolfie = new Wolf();
        Lion leo = new Lion();
        Cat felix = new Cat();
        Dog rover = new Dog();
        ArrayList< Animal > animals = new ArrayList<Animal>();
        animals.add(wolfie);
        animals.add(leo);
        animals.add(felix);
        animals.add(rover);
        for ( Animal a  : animals) {
            a.makeNoise();
            goToBed(a);
        }
    }
}
```

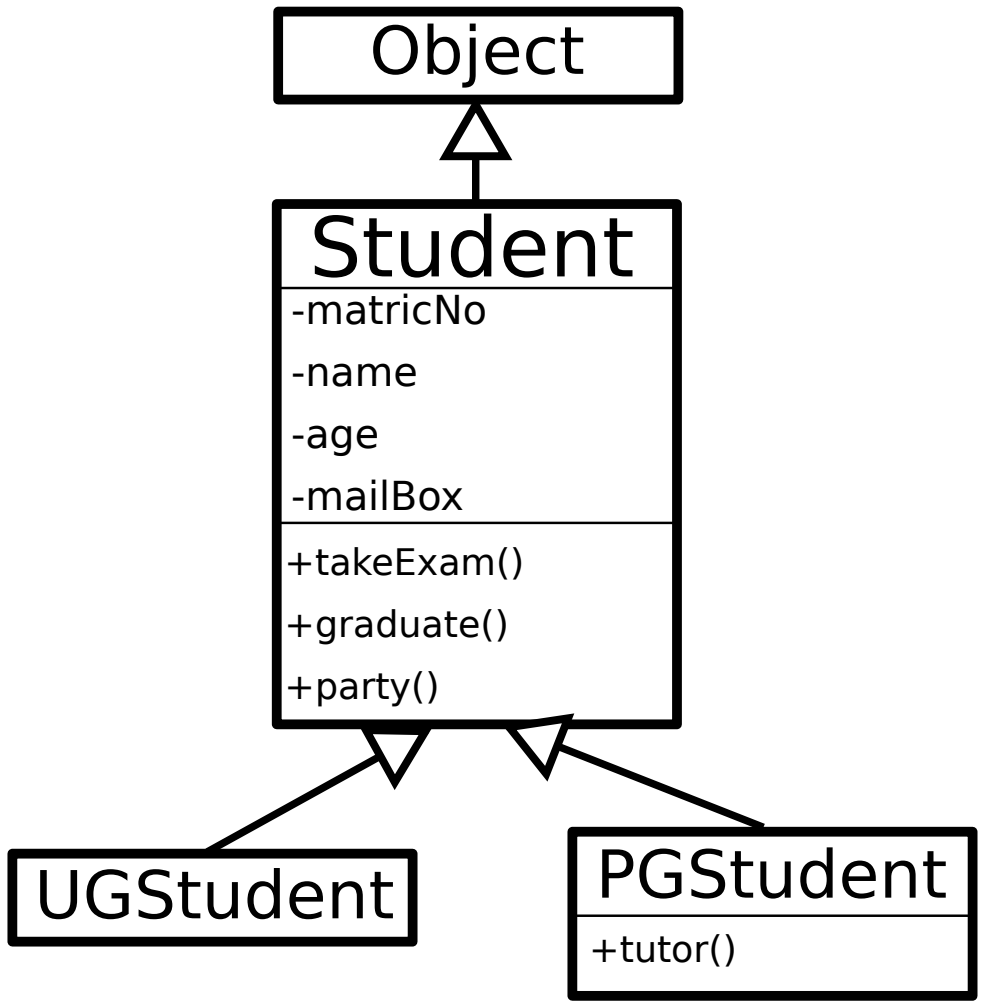
Polymorphic Arrays

`ArrayList<Animal>` is polymorphic.

- `animals.add(wolfie)`
  add an object of type `Wolf`. OK since `Wolf <: Animal`.
- `for (Animal a : animals)`
  for each object a of type `T` such that `T <: Animal` ...
- `a.makeNoise()`
  if a is of type `T`, use `T`'s `makeNoise()` method.
- `goToBed(a)`
  You get at least an Animal, so you can call every method on it an Animal has

Student Hierarchy

Subclass (`UG`, `PG`) inherit from superclass (`Student`) inherits from superclass (`Object`)



Casting Object Types

Does this work?

```
private static void giveTutorial(Student support) {
    support.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```

Casting Object Types

Does this work?

```
private static void giveTutorial(Student support) {
    support.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```

Compiler Error! Student does not have a `tutor()` method

Casting Object Types

Does this work?

```
private static void giveTutorial(Student support) {
     PGStudent pgsupport = (PGStudent) support;
     pgsupport.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```

Casting Object Types

Does this work?

```
private static void giveTutorial(Student support) {
     PGStudent pgsupport = (PGStudent) support;
     pgsupport.tutor();
}

public static void main(String[] args) {
    Student support = new PGStudent();
    giveTutorial(support);
}
```

Yes, I do actually get a PGStudent as argument.

But what if not??

Casting Object Types

Casting Object Types Should be Protected

```
private static void giveTutorial(Student support) {
     if (support instanceof PGStudent) {
         PGStudent pgsupport = (PGStudent) support;
         pgsupport.tutor();
     }
}

public static void main(String[] args) {
    Student support = new UGStudent();
    giveTutorial(support);
}
```

This works and nothing will be printed.

Overriding vs. Overloading

Method Overriding

If a class `C` overrides a method `m` of superclass `D`, ...

For Example

```
public class Animal {
  public Animal findPlaymate() {
     ...
  }
}

public class Wolf extends Animal {

    ???

}
```

Method Overriding

If a class C overrides a method m of superclass D, then:

- Parameter lists must be the same.

```
public class Animal {
  public Animal findPlaymate () {
     ...
  }
}

public class Wolf extends Animal {
  public Animal findPlaymate (int number) { // This is not overriding
    ...
  }
}

public class Wolf extends Animal {
  public Animal findPlaymate () { // This is overriding
    ...
  }
}
```

Method Overriding

If a class C overrides a method m of superclass D, then:

- Parameter lists must be the same.
- The return type must be the same or a subclass of the original.

```
public class Animal {
  public Animal findPlaymate() {
     ...
  }
}

public class Wolf extends Animal {
  public Student findPlaymate() { // This is not overriding
   ...
  }
}

public class Wolf extends Animal {
  public Wolf findPlaymate() { // This is overriding
   ...
  }
}
```

Method Overriding

If a class C overrides a method m of superclass D, then:

- Parameter lists must be the same.
- The return type must be the same or a subclass of the original.
- The overridden method must be at least as accessible as the original.

```
public class Animal {
   protected Animal findPlaymate() {
      ...
   }
}

public class Wolf extends Animal {
   private Student findPlaymate() { // This is not overriding
    ...
   }
}

public class Wolf extends Animal {
   public Wolf findPlaymate() { // This is overriding
    ...
   }
}
```

Method Overriding

If a class `C` overrides a method `m` of superclass `D`, then:

- Parameter lists must be same and return type must be compatible:
  1. signature of `m` in `C` must be same as signature of `m` in `D`; i.e. same name, same parameter list, and
  2. return type `S` of `m` in `C` must such that `S <: T`, where `T` is return type of `m` in `D`.
- `m` must be at least as accessible in `C` as `m` is in `D`

Most versions I showed that did not override, do in fact compile.

Most versions I showed that did not override, do in fact compile.

But they **overload** the method rather than **override** it.

Method Overloading

Overloading: two methods with same name but different parameter lists.

Overloaded `makeNoise`

```
public void makeNoise() {
...
}
public void makeNoise(int volume) {
...
}
```

Overloaded `println`

```
System.out.println(3); // int
System.out.println(3.0); // double
System.out.println((float) 3.0); // cast to float
System.out.println("3.0"); // String
```

Method Overloading

1. Return types can be different.
2. You can't just change the return type — gets treated as an invalid override.
3. Access levels can be varied up or down.

Incorrect override of `makeNoise`

```
public String makeNoise() {
    String howl = "Ouoooooo!";
    return howl;
}
```

```
Exception in thread "main" java.lang.Error:
Unresolved compilation problem:
        The return type is incompatible with Animal.makeNoise()
```

Let's practice that

https://www.theodysseyonline.com/your-brain-is-muscle-exercise-it

Summary

- Inheritance structures can be long and nested.
- Polymorphism is the ability of objects to take on many forms.
  - It allows you to collect various subtypes in the same list, if the list has the supertype parameter.
  - It allows you to use the same client code for different subtypes, if the client code handles the supertype.
- Overriding needs to follow three rules (parameter list, return type, access).
- Otherwise it is likely overloading.
- It is hard to keep an overview if overloading happens accross class hierarchies.

Reading

Objects First

Chapter 10.7 *SubTyping*
Chapter 11 *More About Inheritance*