Introduction to Java (cs2514)

Lecture 8: Interfaces

M. R. C. van Dongen

February 10, 2017

Overloading

- A subclass may *override* a public method from a superclass.
    - Allows subclasses to implement more specific behaviour.
    - E.g. different `rotate( )` behaviour in the `Amoeba` subclass.
- A method's *signature* comprises its name and its argument types.
- Two methods with the same name may also *overload* each other.
    - Has nothing to do with inheritance.
    - The methods must have a different *signature*.
        - E.g. different numbers of arguments.
        - E.g. same number of arguments but at least one different type.
- Class constructors may also overload each other.

Overloading: Example

Java

```
public  void f( int x )            { /* stuff */ }
public  int  f( double x )         { /* stuff */ }
private void f( int x, double y ) { /* stuff */ }
public  void f( double x, int y ) { /* stuff */ }
```

Not Overloading

Same Name and Same Argument Type List

Don’t Try This at Home

```
public  void f( int x ) { /* stuff */ }
private int  f( int x ) { /* stuff */ }
```

Also Not Overloading

Same Name and Same Argument Type List

Don’t Try This at Home

```
public  void f( int x ) { /* stuff */ }
private int  f( int y ) { /* stuff */ }
```

Also Not Overloading

Different Names

Java

```
public  void f( int x ) { /* stuff */ }
private void g( int x ) { /* stuff */ }
```

Constructor Overloading

- Class constructors may also overload each other.
- They may even call each other.
- When they do, you write `this( ... )` for the constructor call.
  - You don’t add `new`.
  - Calling `this( … )` should be the first call in a class constructor.
- Lets you implement easy client-friendly constructors.
  - A very general constructor does the work.
  - The friendly versions call `this( … )`.
  - The friendly version may also do additional configuring.

Example: Constructor Overloading

Java

```java
public class NamedObject {
    private static final String DEFAULT_NAME = "Object";
    private final String name;

    // Default constructor
    public NamedObject( final String name ) {
        this.name = name;
    }

    // Special-purpose constructor constructor
    public NamedObject( ) {
        this( DEFAULT_NAME );
    }
}
```

Motivation for Interfaces

- Let’s assume you have a sorting algorithm.
- The algorithm works for certain kinds of objects.
- Let’s say it works for `Integers`.
- Let’s say you’d like to use the algorithm for `Doubles`.
- Ideally you’d like to *reuse* the algorithm’s implementation.
- But how?

Overloading: How Not To

Don't Try This at Home

```
public int linearSearch( final Integer[] things, final Integer key ) {
    int index = 0;
    while ((index != things.length) && (things[ index ].compareTo( key ) != 0)) {
        index++;
    }
    return (index < numbers.length) ? index : -1;
}
```

Overloading: How Not To

Don't Try This at Home

```
public int linearSearch( final Double[] things, final Double key ) {
    int index = 0;
    while ((index != things.length) && (things[ index ].compareTo( key ) != 0)) {
        index++;
    }
    return (index < numbers.length) ? index : -1;
}
```

Overloading: How Not To

Don't Try This at Home

```
public int linearSearch( final Byte[] things, final Byte key ) {
    int index = 0;
    while ((index != things.length) && (things[ index ].compareTo( key ) != 0)) {
        index++;
    }
    return (index < numbers.length) ? index : -1;
}
```

How To

Java

```
public int linearSearch( final Comparable[] things, final Comparable key ) {
    int index = 0;
    while ((index != things.length) && (things[ index ].compareTo( key ) != 0)) {
        index++;
    }
    return (index < numbers.length) ? index : -1;
}
```

We Need a Contract

- To reuse the method, we need a contract.
- The contract restricts the type of parameter:
  - We must make sure the parameter has the behaviour we need.
- The contract restricts how the parameters may be used:
  - We're only allowed to use certain kinds of instance methods.
- In Java the contract is called an *interface*.
- Using an interface is a multi-stage process;
  1. You *define* the interface (once).
  2. You *implement* the interface (any number of times).

Defining the Interface

- Defining an *interface* is like defining a class.
- You provide the name of the interface.
- You provide the API of the public instance methods.
- You *don't* provide an implementation of the instance methods.

Example

Java

```
public interface Sellable {
    public double getPrice( );               /* No Implementation */
    public void sellTo( final Buyer buyer ); /* No Implementation */
}
```

Implementing the Interface

- Once you’ve defined the interface, you may *implement* it.
- Implementing the interface may be done in any class.
- Implement an interface means defining its public methods.
    - This is called *overriding* the methods.
- Classes may implement as number of interfaces they like.

Example

Java

```
public class Cat implements Sellable {
    private final double price;
    private Buyer owner;

    public Cat( ... ) {
        ...
    }

    @Override
    public double getPrice( ) {
        return price;
    }

    @Override
    public void sellTo( final Buyer buyer ) {
        owner = buyer;
    }
}
```

Example

Java

```java
public class Car implements Sellable {
    private final double price;
    private Buyer owner;

    public Car( ... ) {
        ...
    }

    @Override
    public double getPrice( ) {
        return price;
    }

    @Override
    public void sellTo( final Buyer buyer ) {
        owner = buyer;
    }
}
```

Example

Java

```
public class Bread implements Sellable {
    private final double price;
    private Buyer owner;

    public Bread( ... ) {
        ...
    }

    @Override
    public double getPrice( ) {
        return price;
    }

    @Override
    public void sellTo( final Buyer buyer ) {
        owner = buyer;
    }
}
```

Example

Java

```
public class Soul implements Sellable {
    private final double price;
    private Buyer owner;

    public Soul( ... ) {
        ...
    }

    @Override
    public double getPrice( ) {
        return price;
    }

    @Override
    public void sellTo( final Buyer buyer ) {
        owner = buyer;
    }
}
```

Using the Interface

Java

```
public static void main( Sting[] args ) {
    final Cat cat = new Cat( "Felix" );
    final Car car = new Car( "merc" );
    final Bread pan = new Bread( "white", "chrunchy" );

    final Buyer mary = new Buyer( "Mary" );

    cat.sellTo( mary );
    car.sellTo( mary );
    pan.sellTo( mary );
}
```

Using the Interface

Java

```
public static void main( Sting[] args ) {
    final Soul soul = new Soul( );

    final Buyer devil = new Buyer( "Devil" );

    soul.sellTo( devil );
}
```

Substitution Principle

Interface Version

- □ Let's assume we have an interface Interface.
- □ Let's assume we have a variable Interface var.
- □ At runtime you may assign var any reference to an instance of a class that implements Interface.
- □ More generally, if a class implements Interface you may use its instances when an Interface object reference is expected.
  - □ This is called the *Liskov substitution principle.*
- □ So let's assume the Dog class implements the Animal interface.
- □ Then you can use a Dog when Java expects an Animal.

Java

```
    Animal animal = new Dog( );
```

Substitution Principle

Class Version

- Let’s assume we have an class Clazz.
- Let’s assume we have a variable Clazz var.
- At runtime you may assign var any reference to an instance of a class that extends Clazz.
- More generally, if a class extends Clazz you may use its instances when an Clazz object reference is expected.
  - This is called the *Liskov substitution principle.*
- So let’s assume the Dog class extends the Animal class.
- Then you can use a Dog when Java expects an Animal.

Java

```
    Animal animal = new Dog( );
```

Polymorphism

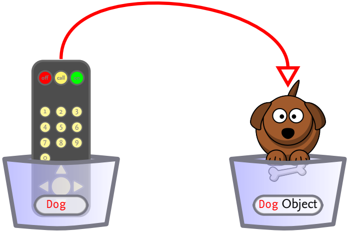
- The term *polymorphism* means
  - *The occurrence of something in several, different forms.*
- A polymorphic reference variable may reference different types of objects over time [Lewis, and Loftus 2009].

Without Polymorphism

- □ The type of reference variable and object are the same:

Java

```
Dog animal = new Dog( );
```

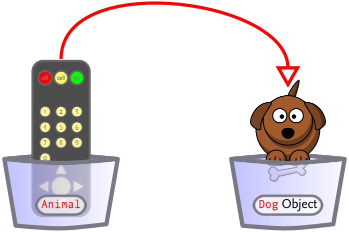


With Polymorphism

- The type of reference variable and object may be different:

Java

```
Animal animal = new Dog( );
```

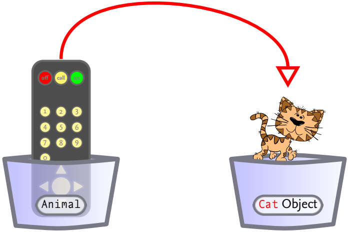


With Polymorphism

- □ The type of reference variable and object :

Java

```
Animal animal = new Cat( );
```



So, With Polymorphism

- The reference type must implement the interface/extend the class.
- *The type of the object, not the type of the reference, determines which instance method is called.*
- This is also known as *late binding*.

Java

```
Animal[] animals = new Animal[ 2 ];
animal[ 0 ] = new Dog( );
animal[ 1 ] = new Sheep( );
animal[ 0 ].makeNoise( ); // Barks
animal[ 1 ].makeNoise( ); // Bleats
```

For a Polymorphic Method Definition

- Formal parameters and return types can be polymorphic.
- With formal parameter `Animal` the actual parameter may be `Dog`.
- Likewise, return type may be `Animal` but a `Cat` may be returned.

Case Study

Java

```
public interface Animal {
    public void makeNoise( ); /* No Implementation */
    ...
}
```

Case Study

Java

```
public class Cat implements Animal {
    ...

    @Override
    public void makeNoise( ) {
        System.out.println( "Mew. Mew." );
    }
}
```

Case Study

Java

```
public class Dog implements Animal {
    ...

    @Override
    public void makeNoise( ) {
        System.out.println( "Arf. Arf." );
    }
}
```

Case Study

Java

```
public class Hippo implements Animal {
    ...

    @Override
    public void makeNoise( ) {
        System.out.println( "Grunt" );
    }
}
```

Case Study

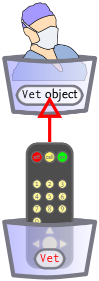
Java

```
public class Vet {
    public void giveShot( Animal animal ) {
        System.out.print( "Giving shot: " );
        animal.makeNoise( );
    }
}
```

Case Study (Continued)

Java

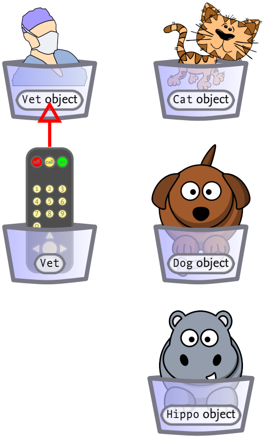
```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```



Case Study (Continued)

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

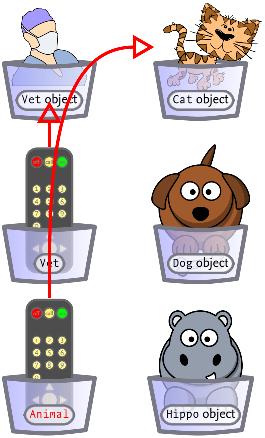


Case Study (Continued)

Iteration #1: animal is a Cat Reference

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

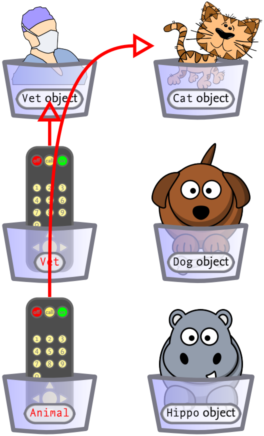


Case Study (Continued)

Iteration #1: Animal expected & Cat implements Animal

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

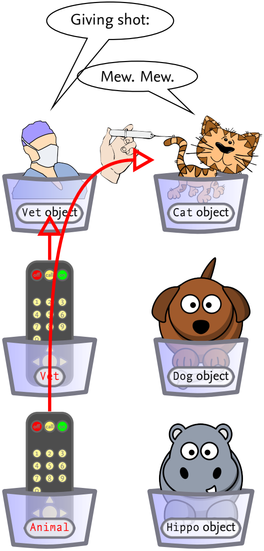


Case Study (Continued)

Iteration #1: vet.giveShot(animal): Use Cat object's makeNoise( )

Java

```java
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

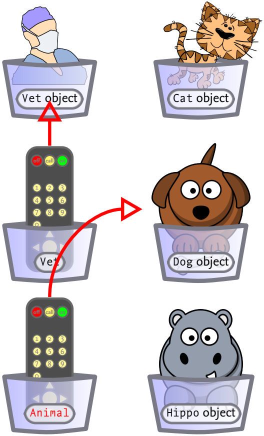


Case Study (Continued)

Iteration #2: animal is a Dog Reference

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

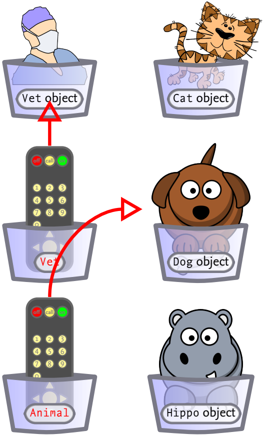


Case Study (Continued)

Iteration #2: `Animal` expected & `Dog` implements `Animal`

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

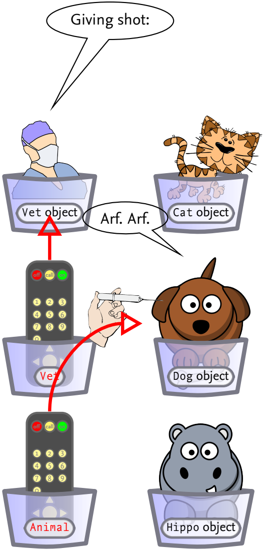


Case Study (Continued)

Iteration #1: vet.giveShot(animal): Use Dog object's makeNoise( )

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

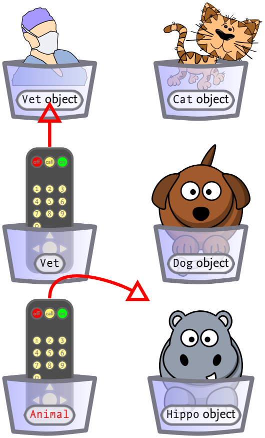


Case Study (Continued)

Iteration #2: animal is a Hippo Reference

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

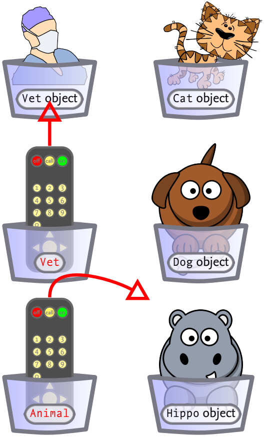


Case Study (Continued)

Iteration #2: Animal expected & Hippo implements Animal

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

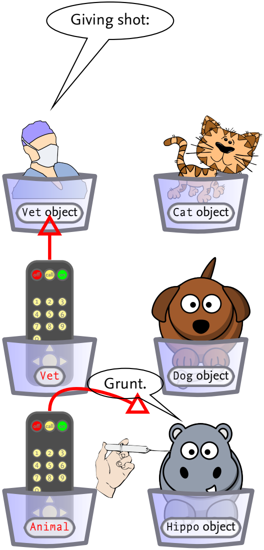


Case Study (Continued)

Iteration #1: `vet.giveShot( animal )`: Use `Hippo` object's `makeNoise( )`

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

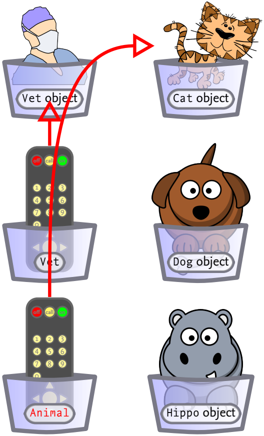


Case Study (Continued)

animal is a Cat Reference

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

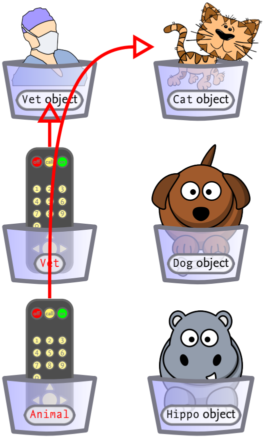


Case Study (Continued)

Animal expected & Cat implements Animal

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

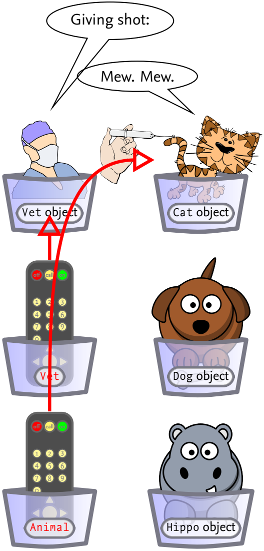


Case Study (Continued)

Use Cat object's makeNoise( ): Giving Shot: Mew. Mew.

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

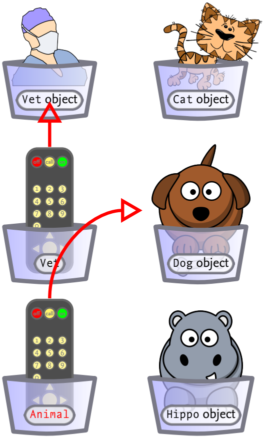


Case Study (Continued)

animal is a Dog Reference

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

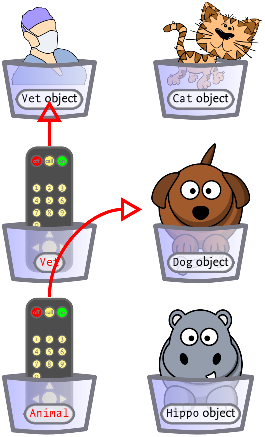


Case Study (Continued)

Animal expected & Dog implements Animal

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```

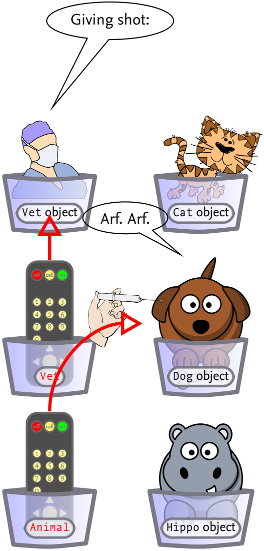


Case Study (Continued)

Use Dog object’s makeNoise( ): Giving Shot: Arf. Arf.

Java

```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal = animals[ 0 ];
    vet.giveShot( animal );
    animal = animals[ 1 ];
    vet.giveShot( animal );
  }
}
```



Delegation

- With interfaces we can simulate inheritance.
- It’s a lot more work but the resulting design may be better:
  - The interface has no implementation.
  - You can depend on a non-exisiting implementation.
- Relies on object compostion rather than inheritance.
  - (Has-a as opposed to is-a.)
- We can re-use existing implementation efforts using *delegation*.
- To implement the interface in class `C`:
  - We need a *concrete* class that implements the interface.
  - `C` implements the interface using a concrete class instance.
  - `C` simply *delegates* the work to the concrete instance.
  - Usually `C` owns the concrete instance.

Example: Delegation

Java

```
public interface Noisy {
    public void makeNoise( );
}
```

Example: Delegation (Continued)

Java

```
public class ConcreteNoisy implements Noisy {
    private final String sound;

    ConcreteNoisy( final String sound ) {
        this.sound = sound;
    }

    @Override
    public void makeNoise( ) {
        System.out.println( sound );
    }
}
```

Example: Delegation (Continued)

Java

```java
public class Dog implements Noisy {
    // We use the polymorphic type @Noisy@, not @ConcreteNoisy@.
    // That way we can only use @Noisy@ behaviour.
    private final Noisy concreteNoisy;

    public Dog( ) {
        concreteNoisy = new ConcreteNoisy( "Arf. Arf." );
    }

    @Override
    public void makeNoise( ) {
        // Here we delegate the noise making
        concreteNoisy.makeNoise( );
    }
}
```

Example: Delegation (Continued)

Notice the (Unavoidable) Code Duplication

Java

```
public class Cat implements Noisy {
    // We use the polymorphic type @Noisy@, not @ConcreteNoisy@.
    // That way we can only use @Noisy@ behaviour.
    private final Noisy concreteNoisy;

    public Cat( ) {
        concreteNoisy = new ConcreteNoisy( "Mew. Mew." );
    }

    @Override
    public void makeNoise( ) {
        // Here we delegate the noise making
        concreteNoisy.makeNoise( );
    }
}
```

- □ A class may only have one direct superclass.
- □ A class may implement any number of interfaces.

Question Time

Questions Anybody?

For Monday

- Study Chapter 7 from the book.
- Study the presentation.
- Read Chapter 8 from the book.

Acknowledgements

- This lecture is partially based on
  - [Sierra, and Bates 2004].

Bibliography I

- Lewis, John, and William Loftus [2009]. *Java Software Solutions Foundations of Program Design*. Pearson International. ISBN: 978-0-321-54934-1.
- Sierra, Kathy, and Bert Bates [2004]. *Head First Java*. O'Reilly. ISBN: 978-0-596-00712-6.

About this Document

- This document was created with `pdflatex`.
- The LaTeX document class is `beamer`.