Software Development (cs2500)

Lecture 20: Interfaces and Polymorphsm

M. R. C. van Dongen

November 6, 2013

Motivation for Interfaces

- Let’s assume you have an algorithm.
- The algorithm works for certain kinds of objects.
- Let’s say it works for numbers.
- Ideally you’d like to *reuse* the algorithm’s implemlementation.
- 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 public instance methods.
- You *don't* provide an implementation of the instance methods.

Example

Java

```
public interface Sellable {
    public double getPrice( );
    public void sellTo( final Buyer buyer );
}
```

Implementing the Interface

- Once you’ve defined the interface, you may *implement* it.
- Implementing the interface may be done in any class.
- To implement the interface you define its public methods.
  - This is called *overriding* the methods.

Example

Java

```
public class Cat {
    ...



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



    ...
}
```

Example

Java

```
public class Car {
    ...



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



    ...
}
```

Example

Java

```
public class Bread {
    ...



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



    ...
}
```

Example

Java

```
public class Soul {
    ...



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



    ...
}
```

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

```
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;



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

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



    @Override
    public void sellTo( final Buyer 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

- 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 if `Interface` 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` if `Java` expects an `Animal`.

Java

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

Polymorphism

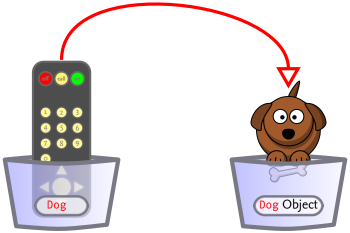
- The term *polymorphism* means
  - *The occurrence of something in several, different forms.*
- A polymorphic reference variable can refer to different types of objects over time [Lewis, and Loftus 2009].

Without Polymorphism

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

Java

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

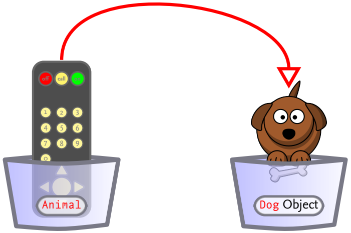


With Polymorphism

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

Java

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

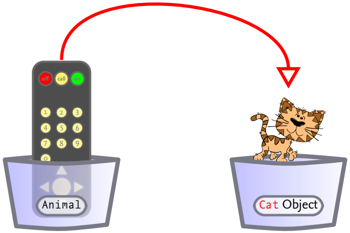


With Polymorphism

- The type of reference variable and object :

Java

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



So, With Polymorphism

- Reference type must implement the interface.
- *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( );
    ...
}
```

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

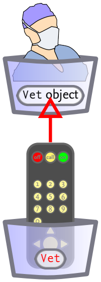
Java

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

Case Study (Continued)

Java

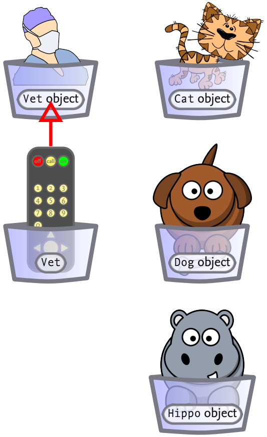
```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal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

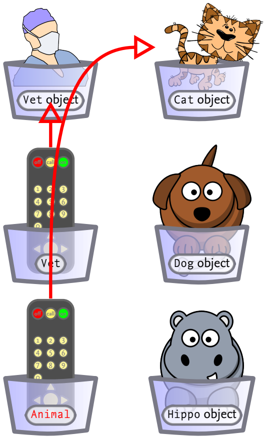


Case Study (Continued)

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 );
  }
}
```

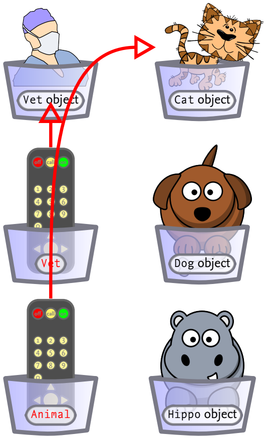


Case Study (Continued)

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 );
  }
}
```

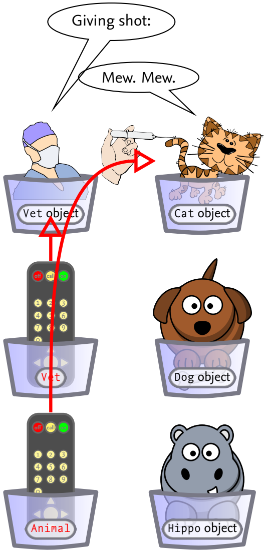


Case Study (Continued)

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

Java

```
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 );
  }
}
```

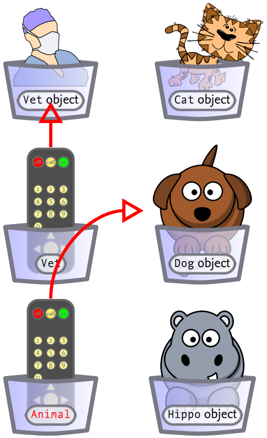


Case Study (Continued)

Iteration #2: `animal` is a `Dog` Reference

Java

```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 );
  }
}
```

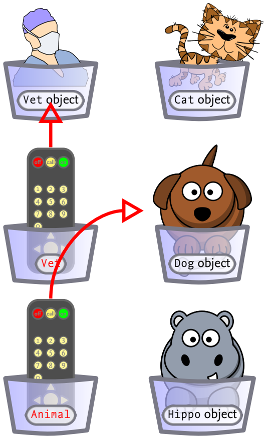


Case Study (Continued)

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

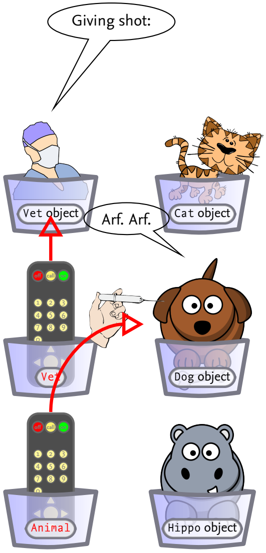
Java

```
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

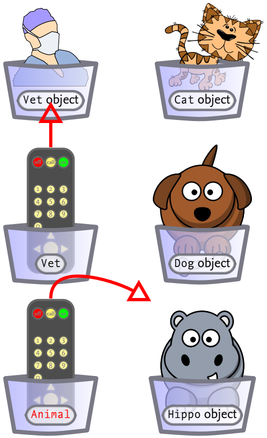
```
public class PetOwner {
  public static void main( String[] args ) {
    Vet vet = new Vet( );
    Animal[] animals = { new Cat( ),
                         new Dog( ),
                         new Hippo( ) };
    for (Animal animal : animals) {
      vet.giveShot( animal );
    }
    Animal animal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

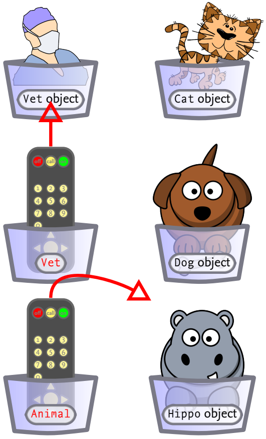


Case Study (Continued)

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 );
  }
}
```

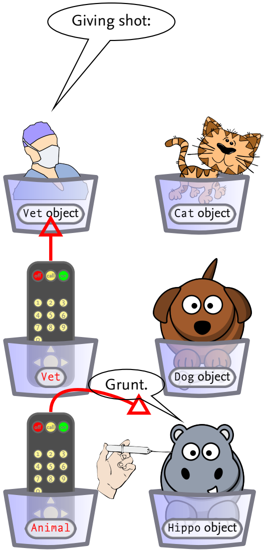


Case Study (Continued)

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 );
  }
}
```

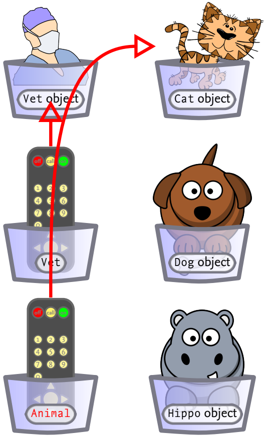


Case Study (Continued)

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 );
  }
}
```

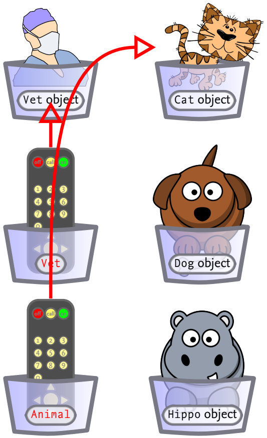


Case Study (Continued)

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 );
  }
}
```

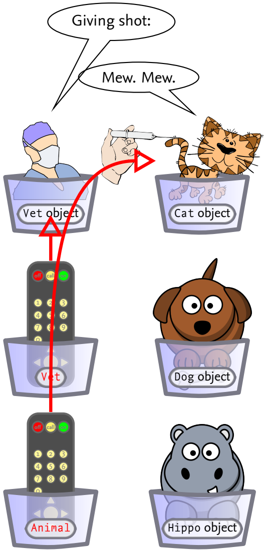


Case Study (Continued)

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 );
  }
}
```

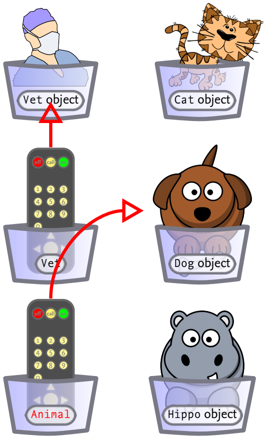


Case Study (Continued)

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 );
  }
}
```

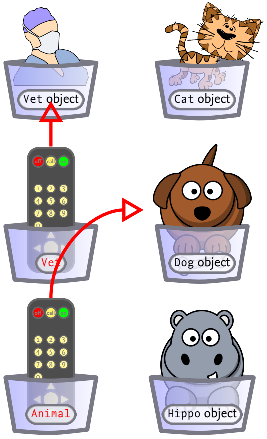


Case Study (Continued)

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 );
  }
}
```

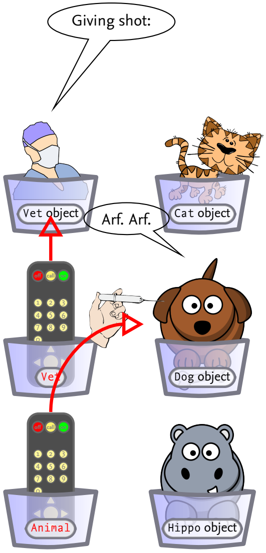


Case Study (Continued)

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 );
  }
}
```



For Friday

- Study [Horstmann 2013, Sections 8.1 and 8.3].

Acknowledgements

- This lecture corresponds to[Horstmann 2013, 8.1–8.3].

About this Document

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