More on Abstraction in Java

Deadly Diamond of Death

Produced by: Mairead Meagher
Dr. Siobhán Drohan

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

Department of Computing and Mathematics
http://www.wit.ie/

RECAP ON ABSTRACTION

Abstract vs Concrete

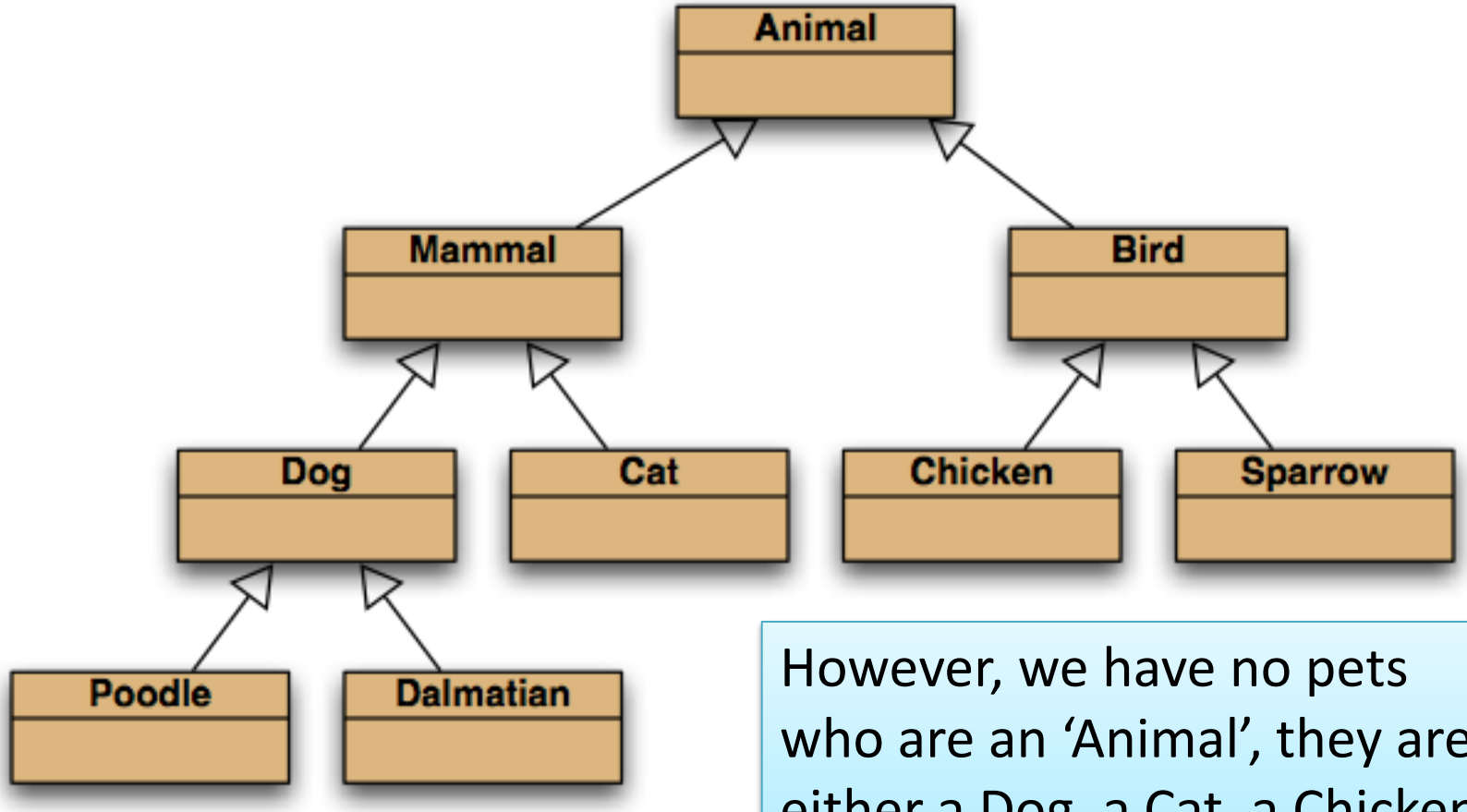
- Abstract
  - Implementation delayed

    → abstract method has no code

    → cannot instantiate an abstract class (it has, by definition “unfinished” methods)

- Concrete
  - Ready to go.
  - Everything up to now has been concrete.

Inheritance hierarchies



However, we have no pets who are an ‘Animal’, they are either a Dog, a Cat, a Chicken, etc.

Abstract Methods

- Abstract methods have `abstract` in the signature.
- Abstract methods have no body.
  - ‘We promise to write this later. Every (concrete) subclass of this class will have this implemented in the subclass.’
- Abstract methods make the class abstract.
  - Think about why this is?

Abstract Classes

- An abstract class is a class that contains zero or more abstract methods.
- An class that has an abstract method must be declared abstract.
- Abstract classes cannot be instantiated.
- Abstract classes function as a “base” for subclasses.

  → abstract classes can be subclassed.
- Concrete subclasses complete the implementation.

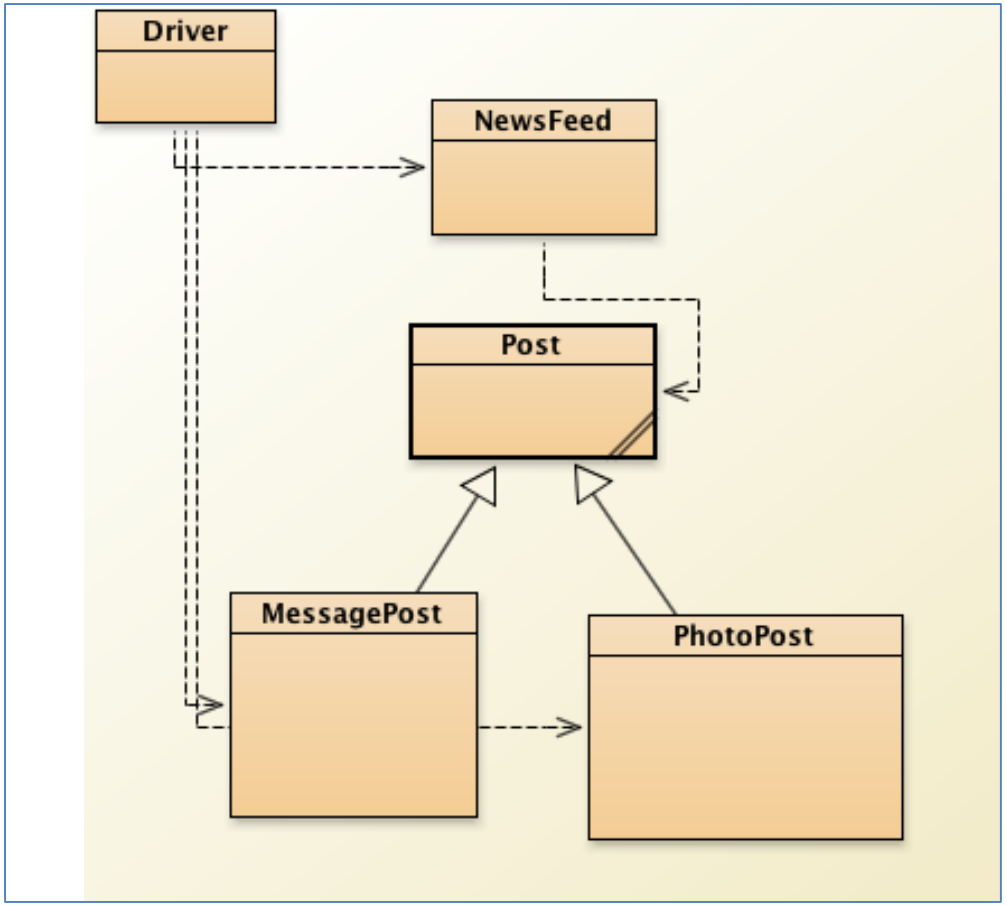
Network-V4 (no abstraction)

```
Options

Posts
1) Add a Text Post
2) Add a Photo Post
3) List all Posts
0) Exit
==>>
```

Our news feed displays either MessagePost or PhotoPost objects.

NOTE: we never create a Post object but our ArrayList is of Post.

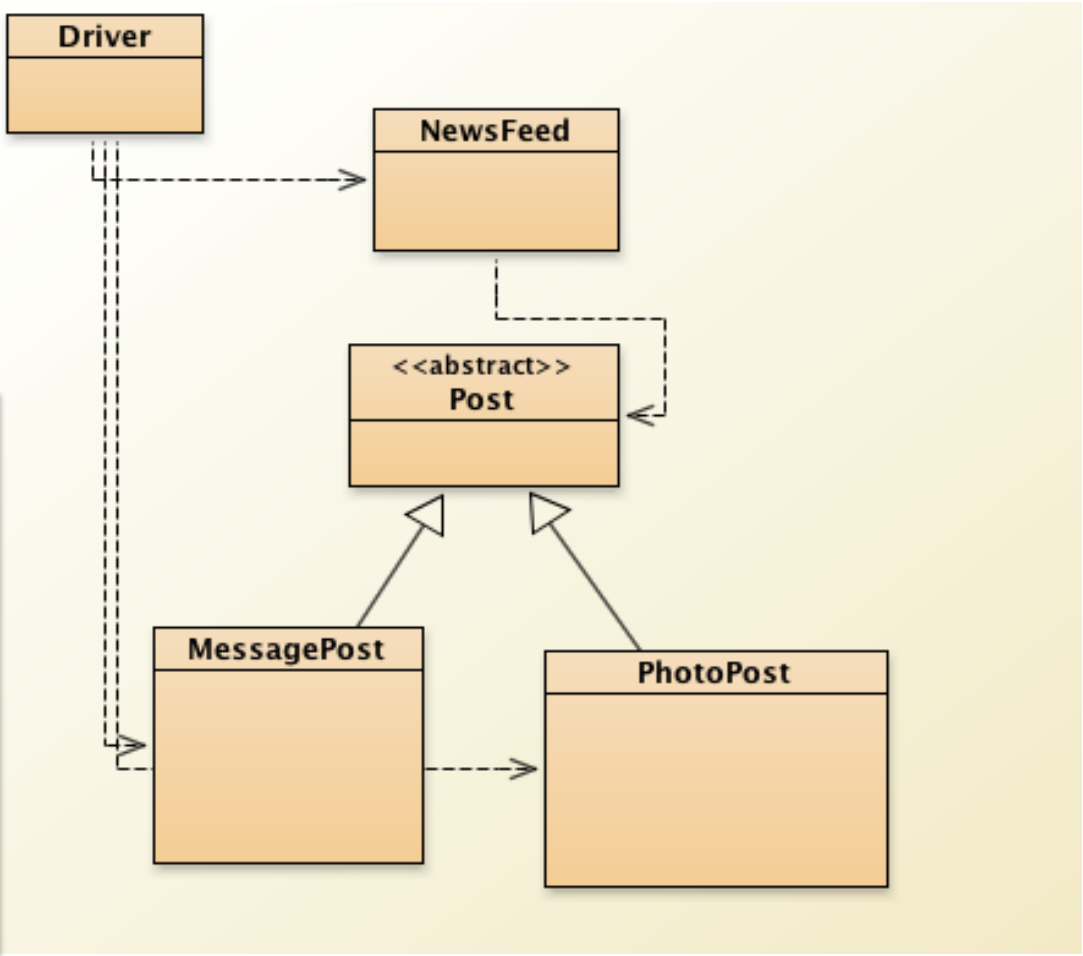


Network-V5 (Post as an abstract class)

```
Options
Posts
1) Add a Text Post
2) Add a Photo Post
3) List all Posts
0) Exit
==>>
```

So, because we never create a Post object but our ArrayList is of Post…

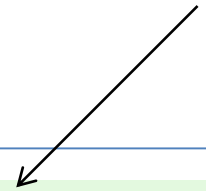
We can make Post abstract!



Network-V5 (Post as an abstract class)

- We can never create a ‘post’ object
  - We cannot instantiate one because Post is abstract.
- In Post, we define fields and methods that can be used later for all subclasses (using super)
  - e.g. display(), constructor.

Syntax for abstract classes



```
public abstract class Post
{
    private String username;  // username of the post's author
    private long timestamp;
    private int likes;
    private ArrayList<String> comments;
```

displayExtract() as an abstract method

- If you wish all subclasses of a class to implement a particular method as part of its code, simply write an abstract method heading in superclass.

- Each subclass must have this method fully coded.

displayExtract() as an abstract method

```
abstract String displayExtract();
```

Post

MessagePost

```
String displayExtract()
{
    return "Message extract "+ message.substring(0,10) + "....";
}
```

PhotoPost

```
String displayExtract()
{
    return "Photo caption: " + caption.substring(0,10) + "....";
}
```

DEADLY DIAMOND OF DEATH

Deadly Diamond of Death!

- Recall that multiple inheritance is not allowed in Java.

- Any idea why the Java designers decided to not allow multiple inheritance?

Deadly Diamond of Death!

- Recall that multiple inheritance is not allowed in Java.
- Any idea why the Java designers decided to not allow multiple inheritance?
- It is because of the Deadly Diamond of Death problem!

Deadly Diamond of Death!

- This is easiest explained by example.

- Let’s pretend that Java allows multiple inheritance and we will see really quickly what the Deadly Diamond of Death is!

http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

- Suppose that we have an abstract super class, with an abstract method in it.

```
public abstract class AbstractSuperClass{
        abstract void do();
}
```

http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

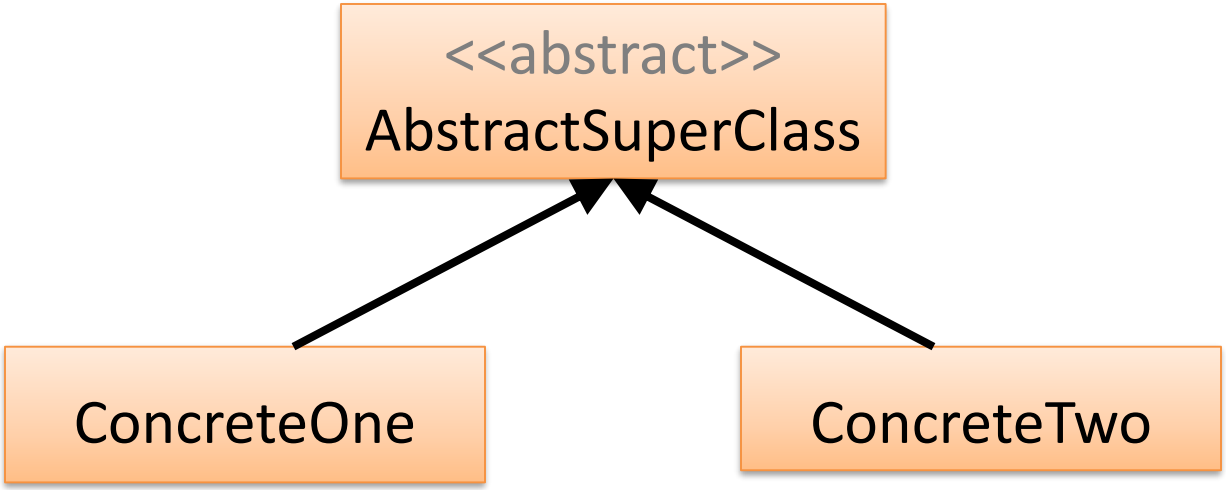
- Now two concrete classes extend this abstract super class.
- Each classes provides their own implementation of the abstract method defined in the super class.

```
public class ConcreteOne extends AbstractSuperClass{
    void do(){
        System.out.println("I am testing multiple Inheritance");
    }
}
```

```
public class ConcreteTwo extends AbstractSuperClass{
     void do(){
         System.out.println("I will cause the Deadly Diamond of Death");
     }
}
```

Deadly Diamond of Death - Example

- So far, our class diagram looks like this:



http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

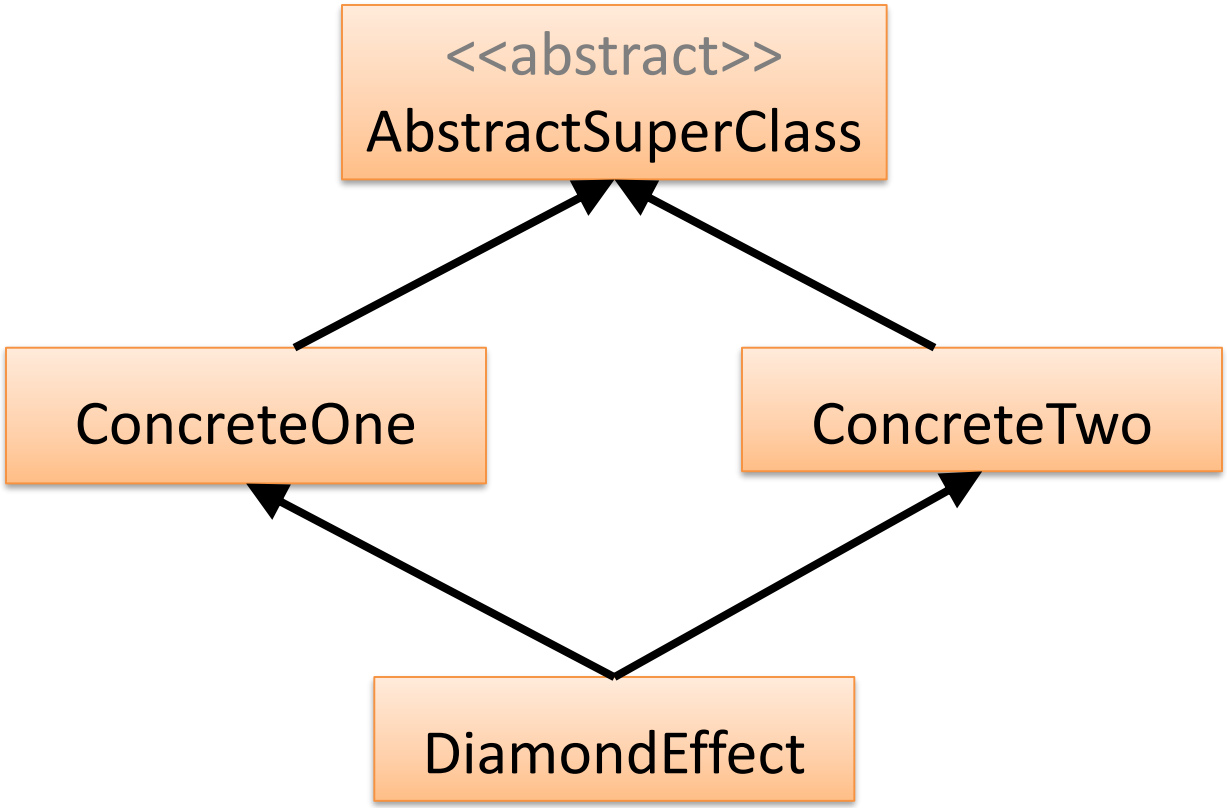
- Now a fourth class comes into picture
  which **extends** the above two concrete classes.

```
public class DiamondEffect extends ConcreteOne, ConcreteTwo{
    //Some methods of this class
}
```

http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

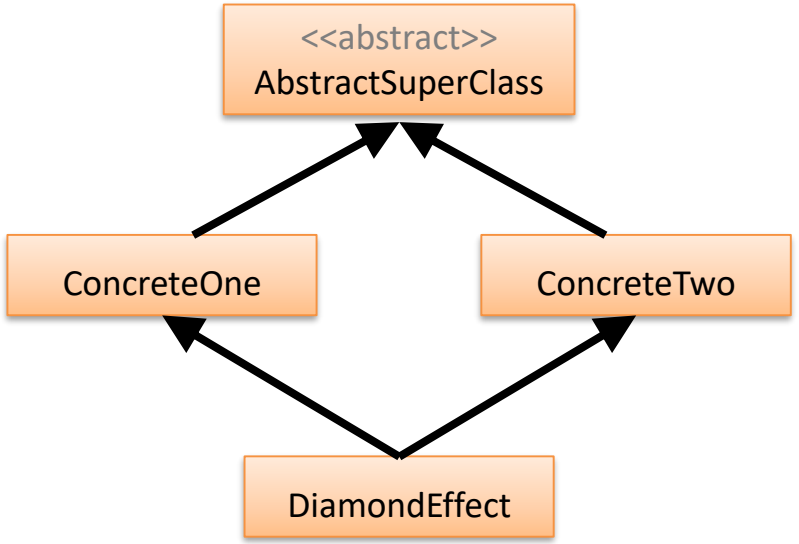
- Note that our class diagram is a diamond shape.



http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

- The DiamondEffect class inherits all the methods of the parent classes.
- BUT we have a common method (void do()) in the two concrete classes, each with a different implementation.
- So which **void do()** implementation will be used for the DiamondEffect class as it inherits both these classes?



http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Deadly Diamond of Death - Example

- Actually no one has got the answer to the above question, and so to avoid this sort of critical issue, Java banned multiple inheritance.

- The class diagram which is formed above is like that of a diamond, but with no solution or outcome, and so it is called Deadly Diamond of Death.

http://javacodeonline.blogspot.ie/2009/08/deadly-diamond-of-death.html

Any Questions?