CM 10227: Lecture 8

Dr Rachid Hourizi and Dr Michael Wright

November 22, 2016

Resources

- More help with this course
  - Moodle
  - E-mail - programming1@lists.bath.ac.uk
- Online C and Java IDE
  - https://www.codechef.com/ide
  - Remember to select Java from the drop down menu.
- Books
  - Java by Dissection (Free pdf online)
  - The Java Tutorial (https://docs.oracle.com/javase/tutorial/)

Resources

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - A labs (Continued from week 1)
  - B labs
  - ... Wednesday 11:15-13:05 EB0.7
  - ... Fridays 17:15 to 19:15 in CB 5.13)
  - The Drop in Session
  - ... booked 20 min appointments
  - ... Friday 11.15-13.05 1E 3.9
  - PAL sessions (Mondays 14:15 to 15:05 1E 3.9)

Last Week

- Recap on Classes and Objects
- Inheritance
- Polymorphism

This Week

- Interfaces
- Abstract Classes

Recap: Inheritance and Polymorphism

- Last week we looked at inheritance

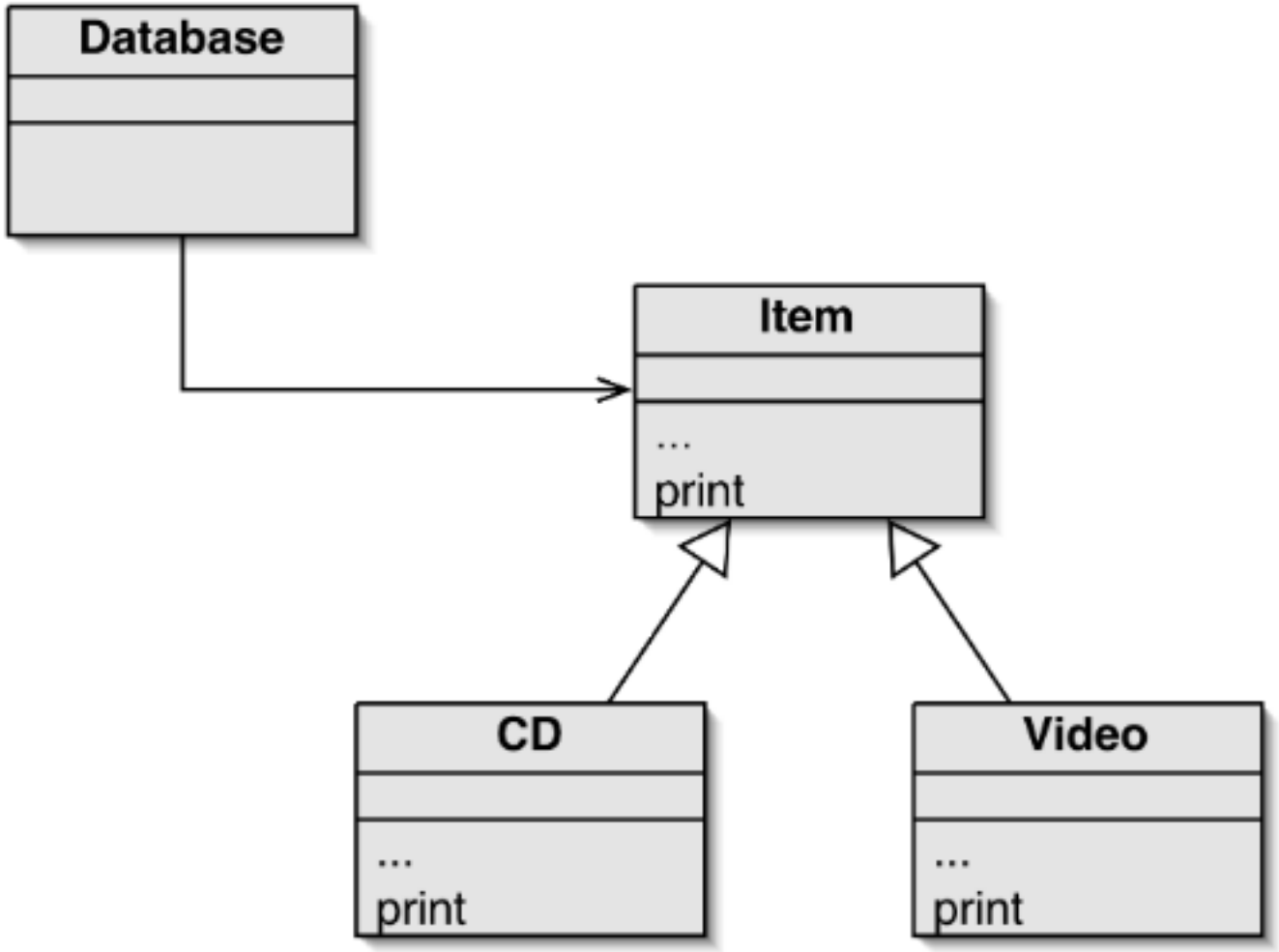
Database Of Multimedia Entertainment

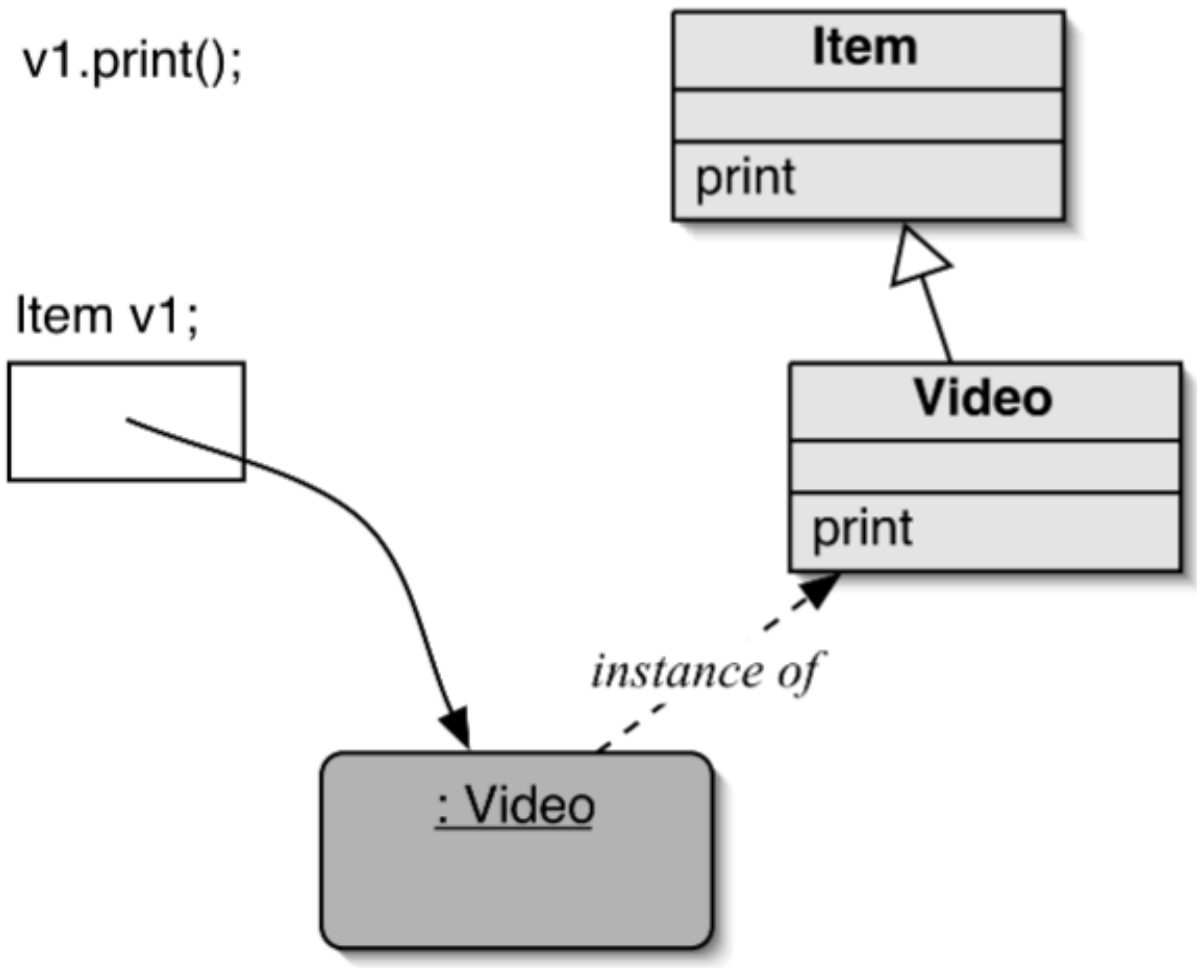
```
public class Item{
    private String title ;
    private int playingTime ;
    private boolean gotIt ;
    private String comment;

    // constructors and methods omitted ...
}

public class MusicFile extends Item{
    private String artist;
    private int numberOfTracks;
    // constructors and methods omitted
}
```

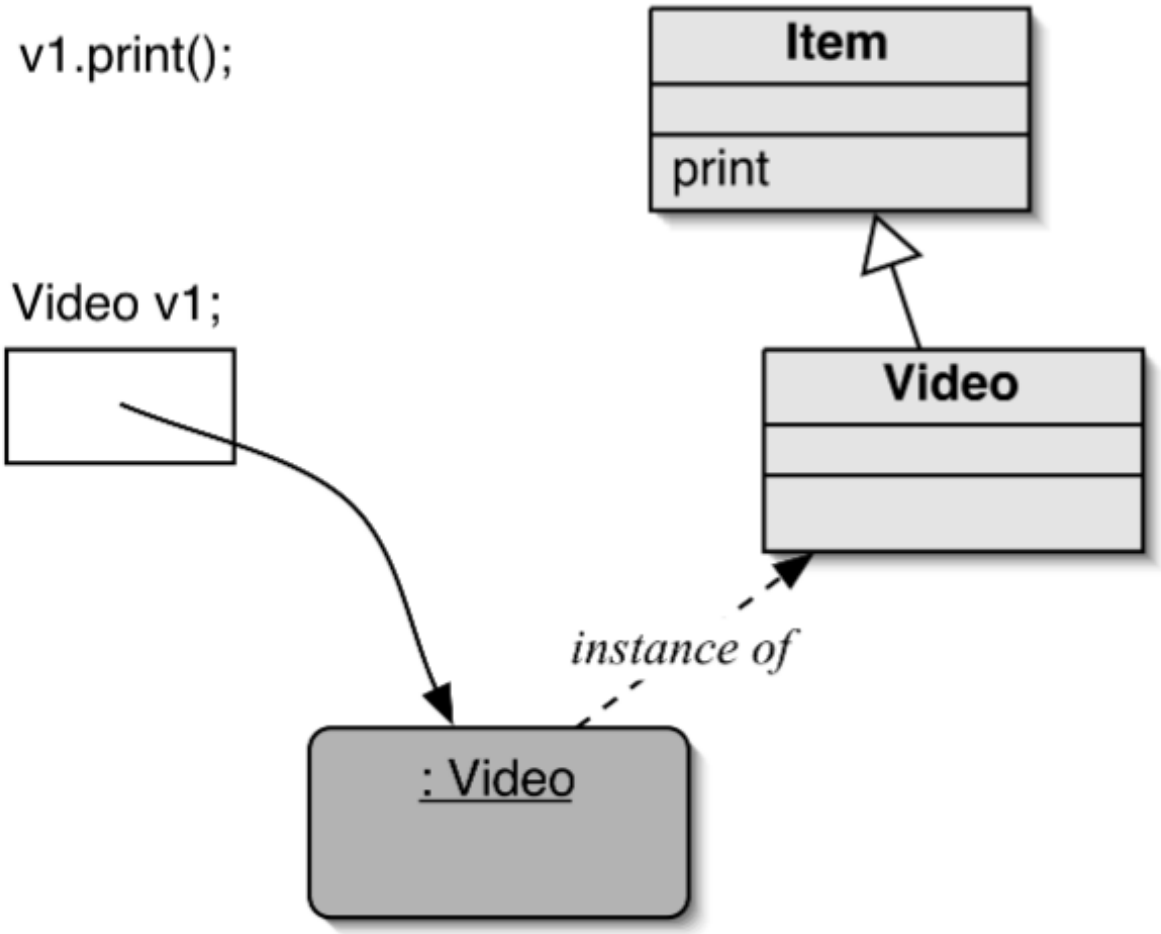
Database Of Multimedia Entertainment (... of my youth)

```
public class Item{
    private String title ;
    private int playingTime ;
    private boolean gotIt ;
    private String comment;

    // constructors and methods omitted ...
}

public class CD extends Item{
    private String artist;
    private int numberOfTracks;
    // constructors and methods omitted
}
```

- Looked at the concept of overridding methods
- i.e. extending or rewriting methods in subclasses when they already exist in superclasses


Database
Item
...
print
CD
...
print
Video
...
print


v1.print();
Item
print
Item v1;
Video
print
instance of
: Video


v1.print();
Item
print
Video v1;
Video
instance of
: Video

Question

- Does it make sense in out Database of Multimedia Entertainment to be able to instantiate Item objects?
- i.e. is adding an Item to our database meaningful?

Question

- Does it make sense in out Database of Multimedia Entertainment to be able to instantiate Item objects?
- i.e. is adding an Item to our database meaningful?

- Not really...
- ... it would be better to use an Interface

Interfaces

- At some point we may want an even looser relationship between superclass and subclass
- We may want to specify a superclass in which no methods are implemented...
- ... leaving subclasses to do all of the implementation

- For example, we might want to describe the methods provided by all classes that can be added to our DOME
- ... and not specify anything about the way these items are printed
- ... i.e. we want all the implementation described elsewhere

- We might specify the Item interface

- In Java, this "specify but implement nothing" approach is achieved through the use of Interfaces
- ... use the **implements** keyword in the classes we define

- Item interface

```
interface Item{
    void print();
}

public class CD implements Item{
     private String title;

     public void print(){
          System.out.println(this.title);
     }
}
```

- A Java interface is a specification of a type
- ... in the form of a type name and methods
- ... that does not define any implementation of methods

- A Java interface...
- ... uses **interface** instead of **class**
- ... all methods are public (no need to mention)
- ... all methods are abstract i.e. contains no implementation (again no need to mention)
- ... no constructors
- ... only constant fields are allowed - **public static final**

Aside : static and final

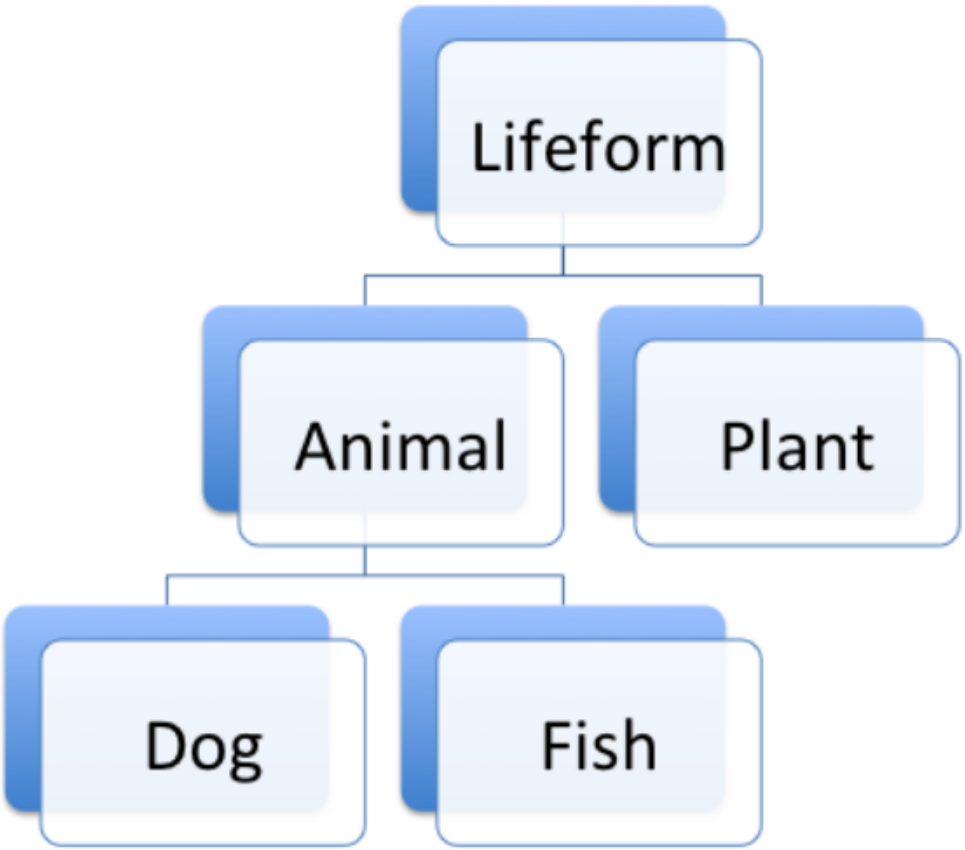
- **static**

- means that the variable or method is shared between all instances of that class
- it belongs to the type, not the actual objects themselves
- e.g. if you have a variable: **private static int i = 0;**
- and you increment it in one instance - **i++;**
- the change will be recreated in all instances

Aside : static and final

- **final**

- you cannot change the value of final variable, method or class
- for a variable it is constant
- for a method you can not override it
- for a class you can not extend it


Lifeform
Animal
Plant
Dog
Fish

- Animal interface

```
interface Animal{
    public static final int CONSTANT_VARIABLE = 42;

    String makeSound();

}
```

- Classes implement an interface

```
public class Fox implements Animal{
    public String makeSound(){
        return "Ring-ding-ding-ding-dingeringeding!";
    }
}
```

- Important to note that Interfaces are not classes
- ... they can not be instantiated
- ... they cannot share a name with a class
- ... cannot contain implemented methods
- ... can only contain method stubs and constants

- Interfaces are more properly described as **design patterns**
- They are, therefore, free of some of the restrictions applied to classes
- You can, for example implement two interfaces in a single subclass

```
public class Fox implements Lifeform, Animal{
    ...
}

public class Tree implements Lifeform, Plant{
    ...
}
```

Aside : Multiple Inheritance

- This is the closest that Java comes to allowing two superclasses (parents) in an inheritance heirarchy
- Subclasses can extend a superclass and implement one or more interfaces
- Or simply implement one or more interfaces

Aside : Multiple Inheritance

- Having a class inherit directly from multiple ancestors.
- Each object oriented programming language has its own rules.
- Java forbids it for classes.
- Java permits it for interfaces.

- Why? (answer later on...)

Back to Interfaces

- Classes implementing an Interface do not inherit code, but ...
- ... are subtypes of the interface type.
- So, polymorphism is available with interfaces as well as classes.

```
Animal fox = new Fox();

Item item = new CD();
Item item = new Video();
```

Bringing this all together...

- Interfaces should be used if you

- Know what to do but don't know how to do it
- Expect unrelated classes to implement your interface
- Want to specify the behaviour of a particular data type...
- ... but are not concerned about who implements its behaviour
- You want to take advantage of multiple inheritance of type

Bringing this all together...

- For example,

- ... consider methods of payment accepted in your application
- ... we know we need to take payment but there are different ways of paying
- ... e.g. Credit Card, PayPal etc..

```
interface Payment{
    void makePayment(double debit);
}
```

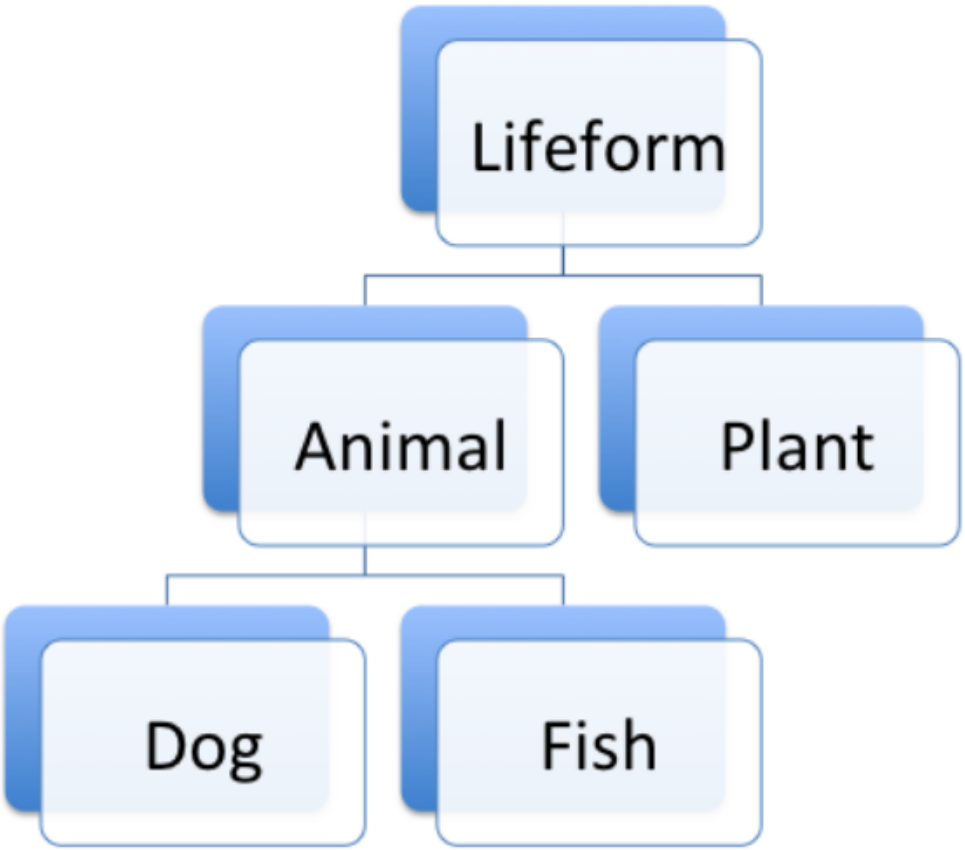
```
public class PayPal implements Payment{
    public void makePayment(double debit) {
        // logic for PayPal payment
        // e.g. Paypal uses username and password for
            payment
    }
}
public class CreditCard implements Payment{
    public void makePayment(double debit){
        // logic for CreditCard payment
        // e.g. CreditCard uses card number, date of
            expiry etc...
    }
}
```

```
public class NozamaUserDetails{

    private String name;
    private Payment paymentMethod;

    public NozamaUserDetails(String name, Payment
        paymentMethod){
         this.name = name;
         this.paymentMethod = paymentMethod;
    }

    // other methods...
}
```

```
public class NozamaRegisterUser{

    public NozamaUserDetails registerUser(String name){
          Payment p = getPaymentMethod();
          return new NozamaUserDetails(name,
              paymentMethod);
     }

     private Payment getPaymentMethod(){
          return new PayPal();
     }
}
```

```
public class ProcessPayment{

    public void purchase(NozamaUserDetails user, Item
        item){
         Payment p = user.getPaymentMethod();
         p.makePayment(item.getPrice());
    }

    // other methods...
}
```

Abstract Classes


Lifeform
Animal
Plant
Dog
Fish

- In some situations, however, we may want to include methods in superclasses but only allow them to be used by subclasses
- For example, in the Animal class we might want to include common functionality for all animals
- ... such as average life span

- ... we may also want to include an makeSound() method in the Animal class
- ... but only allow it to be used in classes that describe specific kinds of animal (i.e. in subclasses of Animal)

- In these cases, we can use abstract classes and methods

```
public abstract class Animal{
    private int averageAge;

    public Animal(int averageAge){
        this.averageAge = averageAge;
    }

    public int getAverageAge(){
        return averageAge;
    }

    // more methods

    // Make the sound of this animal
    abstract void makeSound();
}
```

```
public class Dog extends Animal{

    private String sound;

    public Dog(int averageAge, String sound){
        super(averageAge);
        this.sound = sound;
    }

    // Make the sound of this animal
    public void makeSound(){
        System.out.println("The dog goes " + sound);
    }
}
```

- Abstract methods have abstract in the signature
- Abstract methods have no body
- Abstract methods make the class abstract
- **Abstract classes cannot be instantiated**
- Concrete subclasses complete the implementation

- Note that some methods can still be written in full in an abstract class ('implemented' methods)

- So an abstract class can have
  - constants
  - fields
  - abstract methods
  - implemented methods

- Also note that, like Interfaces, Abstract Classes can not be instantiated
- Classes implementing an Abstract Class are subtypes of the Abstract Class type.
- So, polymorphism is available with Abstract Class as well as classes.

Bringing this all together...

- For example,

- ... consider methods of payment accepted in your application
- ... we know we need to take payment but there are different ways of paying
- ... e.g. Credit Card, PayPal etc..

Bringing this all together...

- BUT, if for example...
- ... each payment type (PayPal, Credit Card etc.) needs to be authorised by a Bank
- ... and this process is the same for all types of payment
- ... and we don't want programmers to reimplement this each time

- ... we would use an Abstract Class to implement this method
- ... but ensure that programmers implement the makePayment method
- ... because this is specific to each payment type

```
public abstract class Payment{

    protected boolean authoriseWithBank(String
        userAuthKey){
        // logic to authorise payment with bank
    }

    abstract void makePayment(double debit);
}
```

```
public class PayPal extends Payment{
    public void makePayment(double debit) {
        // logic for PayPal payment
        // e.g. Paypal uses username and password for
            payment
        super.authoriseWithBank(authToken);
    }
}
public class CreditCard extends Payment{
    public void makePayment(double debit){
        // logic for CreditCard payment
        // e.g. CreditCard uses card number, date of
            expiry etc...
        super.authoriseWithBank(authToken);
    }
}
```

```
public class NozamaUserDetails{

    private String name;
    private Payment paymentMethod;

    public NozamaUserDetails(String name, Payment
        paymentMethod){
         this.name = name;
         this.paymentMethod = paymentMethod;
    }

    // other methods...
}
```

```
public class NozamaRegisterUser{

    public NozamaUserDetails registerUser(String name){
          Payment p = getPaymentMethod();
          return new NozamaUserDetails(name,
              paymentMethod);
     }

     private Payment getPaymentMethod(){
          return new PayPal();
     }
}
```

```
public class ProcessPayment{

    public void purchase(NozamaUserDetails user, Item
        item){
         Payment p = user.getPaymentMethod();
         p.makePayment(item.getPrice());
    }

    // other methods...
}
```

- Abstract classes should be used if you

- Want to share code among several closely related classes
- Expect that classes that extend your abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private)
- Want to declare non-static or non- final fields (interfaces). This enables you to define methods that can access and modify the state of the object to which they belong

- Note, however, that abstract classes are still classes
- ... more specifically, partially implemented classes

- So the rules that govern implemented classes also govern abstract classes
- i.e. no subclass can have two (or more) abstract parents

Aside : Multiple Inheritance

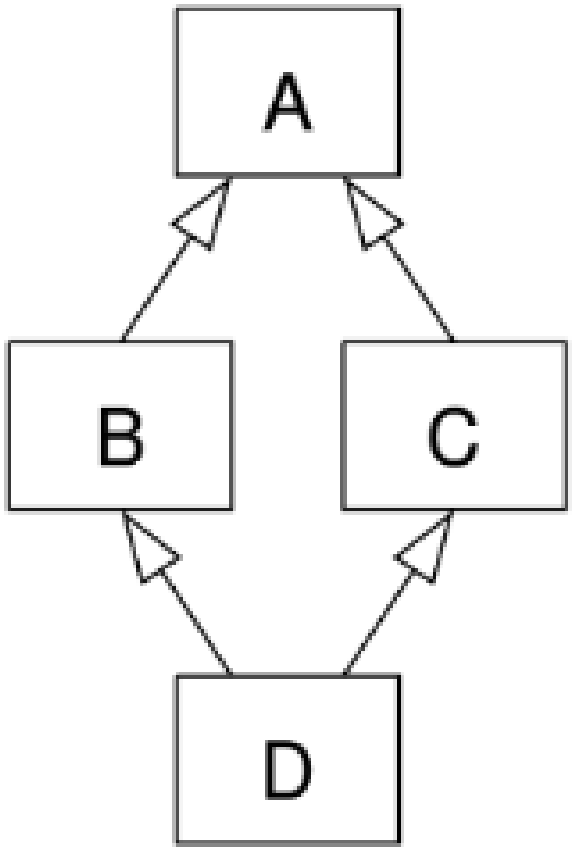
- Having a class inherit directly from multiple ancestors.
- Each object oriented programming language has its own rules.
- Java forbids it for classes.
- Java permits it for interfaces.

- Why? (answer later on...)

Aside : Multiple Inheritance

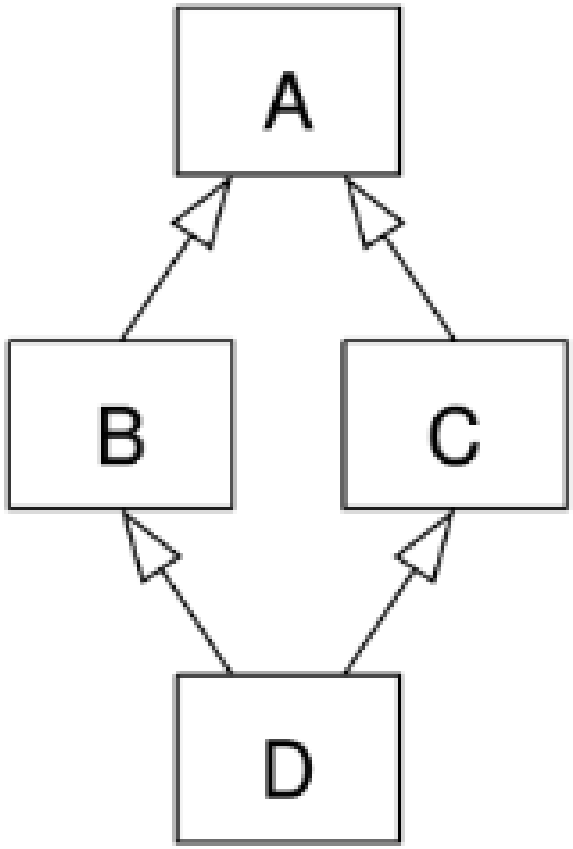
- Answer is - avoids competing definitions of methods or fields?

Dimond Problem



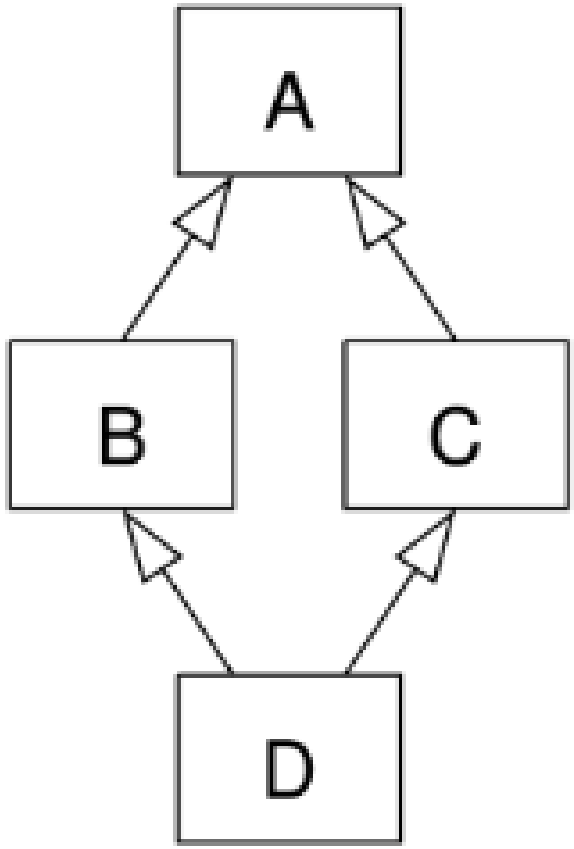
Dimond Problem

- Two classes B and C
- ... inherit from A



Aside : Multiple Inheritance

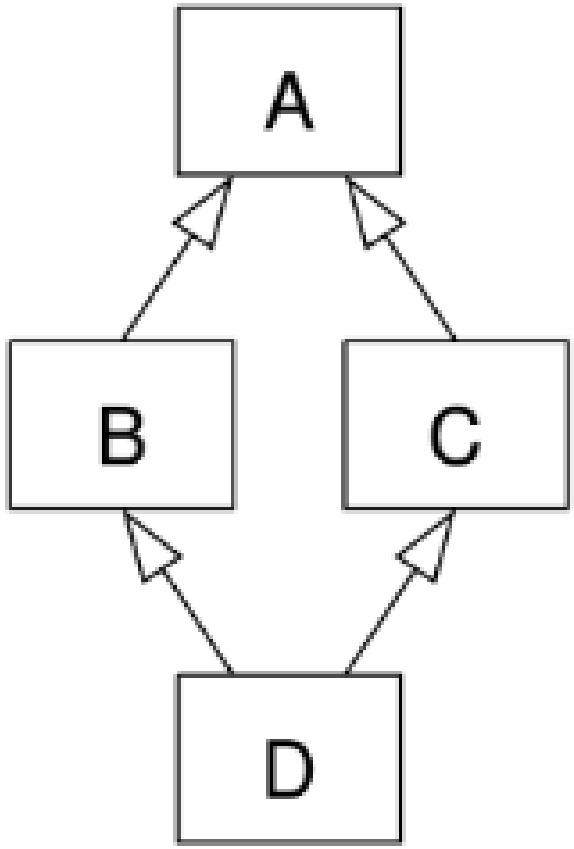
- ... another class D
- ... inherits from both B and C



Aside : Multiple Inheritance

- If there is a method in A
- ... that B and/or C has overridden
- ... and D does not override it

- Then which version of the method does D inherit?
- ... that of B, or that of C?



Aside : Multiple Inheritance

```
public class Book {
    public void getContent(){
        return content;
    }
}
```

Aside : Multiple Inheritance

```
public class AudioBook extends Book{
    public void getContent(){
        return audioContent;
    }
}


public class EBook extends Book {
    public void getContent(){
        return ebookContent;
    }
 }
```

Aside : Multiple Inheritance

```
public class MultiMediaBook extends AudioBook, EBook{

    // other methods BUT NOT getContent()!

    public static void main(String[] args){
        MultiMediaBook mmb = new MultiMediaBook();
        mmb.getContent();
        // which getContent do we use?
        // AudioBook or EBook?
    }
}
```

Aside : Multiple Inheritance

- Java forbids it for classes.
- Java permits it for interfaces.
- So, there are no competing implementations.

- For interest: Java 8 introduces default methods for interfaces which can lead to the diamond problem
- Therefore, the Java compiler provides rules to determine which default method a particular class uses

Interfaces vs Abstract Classes

- It can be difficult to identify
  - when to use an abstract class
  - when to use an interface
- As a simple rule of thumb, when faced with a choice between abstract classes and interfaces
  - use an abstract class when
  - you want to implement some but not all of a class's methods
  - and you are willing to accept the restrictions imposed upon classes
  - e.g. single inheritance
  - otherwise use an interface

Summary

- Inheritance can provide shared implementation...
- ... both as concrete and abstract classes
- Inheritance provides shared type information
- ... interfaces

Summary

- Abstract classes function as incomplete superclasses
- Interfaces provide specification without implementation

- Both Interfaces and Abstract Classes support polymorphism