CM 10227: Lecture 6

Dr. Rachid Hourizi and Dr. Michael Wright

November 12, 2015

Resources

- More help with this course
  - Moodle `http://moodle.bath.ac.uk/course/view.php?id=30475`
  - E-mail - programming1@lists.bath.ac.uk
- Online Java IDE
  - `https://www.codechef.com/ide`
  - Remember to select Java as the language you are coding in...
- General Help on Java
  - How to Think Like a Computer Scientist (Java Version)
  - `http://www.greenteapress.com/thinkjava/thinkjava.pdf`
  - The Java Tutorial (http://docs.oracle.com/javase/tutorial)

- The places that you can get additional support if you are finding the pace of the course a little fast now include
  - The A and B labs
  - The Drop in Sessions

- If you struggling with the exercises, pace of the course and/or coding in general
- Please come and see Rachid or Michael

- If you are finding the pace a little slow on the other hand,
- You can sign up for the Advanced Programming Labs

- When and Where
  - Friday 12.15 - 14.15
  - 1WN 3.12

Last time...

- First Classes and Objects

This week

- Recap on Classes and Objects

- Inheritance
- Polymorphism
- Abstract Classes
- Interfaces

Classes and Objects

- First Objects (In Java)
- Interacting Objects (In Java)

- *Variables*
- Constructors
- Accessors and Mutators
- Methods
- public and private key words

```
public class Example{

}
```

```
public class Example{
        private int exampleVariable;
}
```

```
public class Example{
        private int exampleVariable;

        public Example(){
                exampleVariable = 1;
        }
}
```

```
public class Example{
        private int exampleVariable;

        public Example(){
                exampleVariable = 1;
        }

        public Example(int value){
                exampleVariable = value;
        }
}
```

```
public class Example{
        private int exampleVariable;

        public Example(){
                exampleVariable = 1;
        }

        public Example(int value){
                exampleVariable = value;
        }

        public void setExampleVariable(int value){
                exampleVariable = value;
        }
}
```

```
public class Example{
        private int exampleVariable;

        public Example(){
                exampleVariable = 1;
        }

        public Example(int value){
                exampleVariable = value;
        }

        public void setExampleVariable(int value){
                exampleVariable = value;
        }

        public int getExampleVariable(){
                return exampleVariable;
        }
}
```

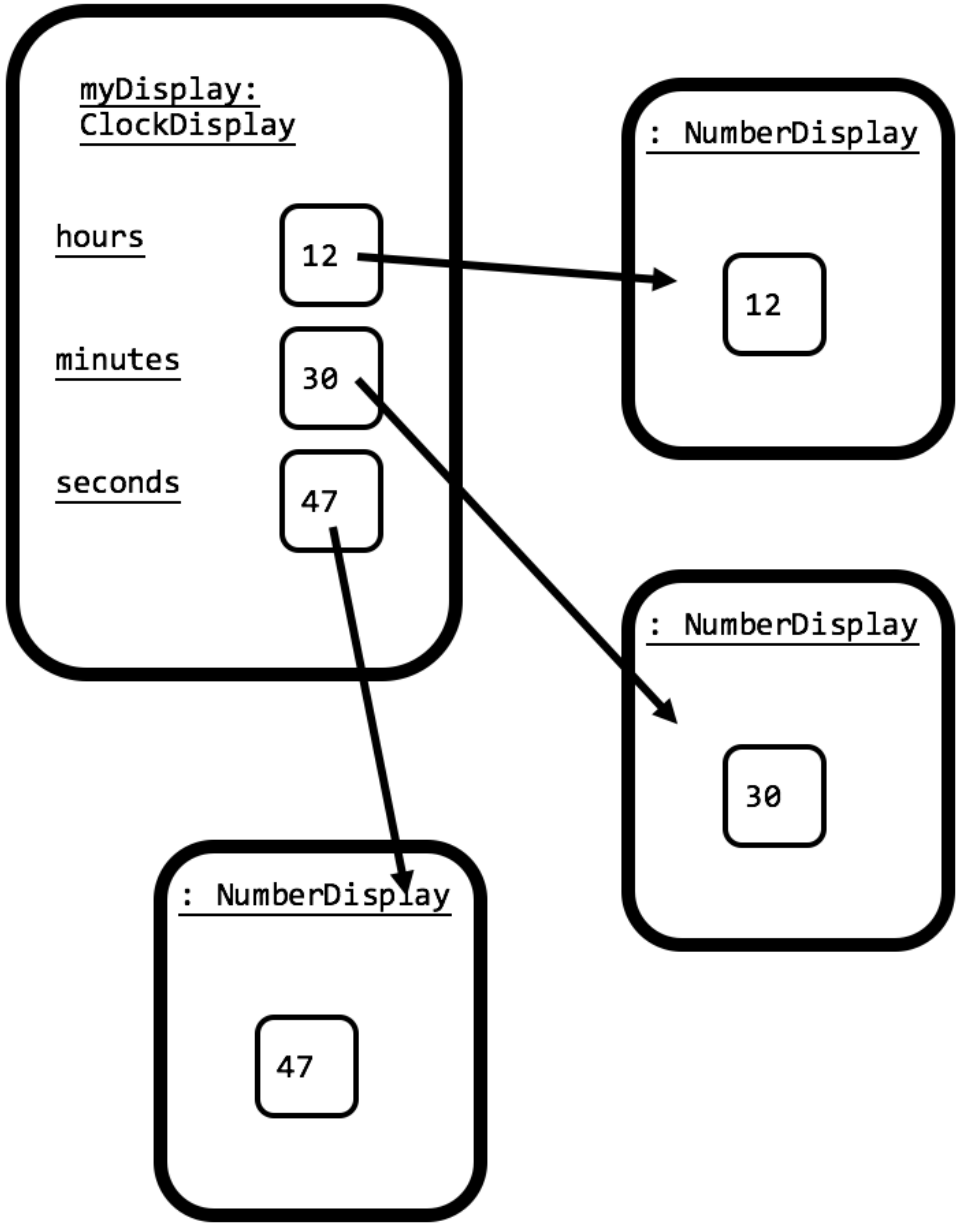
Developing Interactive Classes : A Digital Clock

Question : should we develop one 6 digit display or three 2 digit displays?

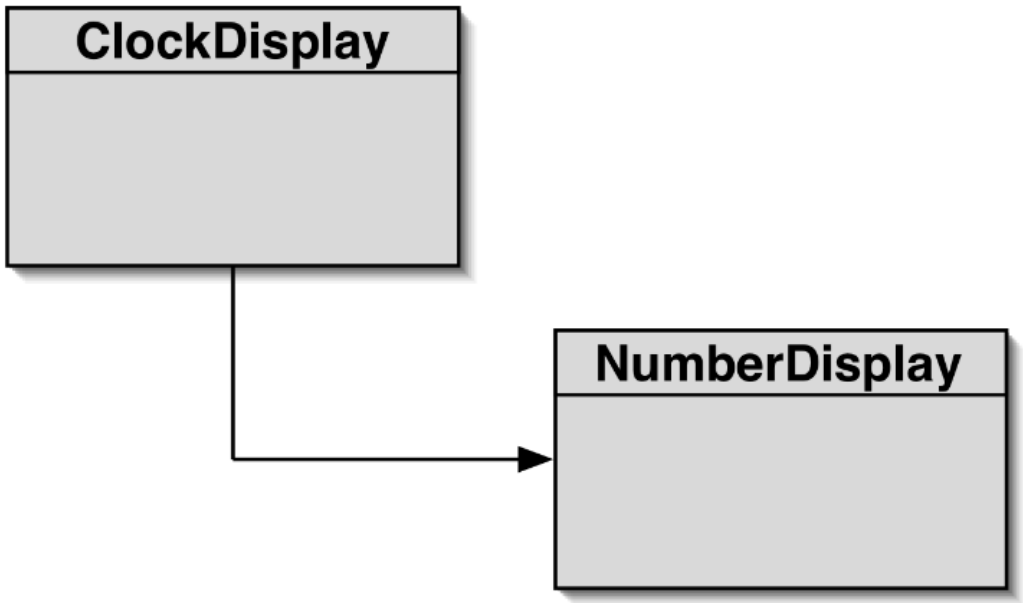
- As we go through this example, we will consider challenges of abstraction and modularization:
  - Abstraction is the ability to ignore details of parts to focus attention on a higher level of a problem.
  - Modularization is the process of dividing a whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.

- A sensible approach if we go for 3*2-digit displays ('NumberDisplay') is to
  - write a NumberDisplay class which allows us to show any two digit number (i.e. a template for all NumberDisplay)
  - write a ClockDisplay class that creates three NumberDisplay instances
  - In other words, develop one class
  - that (in turn) creates three instances of another class (three Objects)

Object diagram



Class diagram



- Class Diagrams
  - Show the classes of an application and the relationships between them
  - Give information about the source code
  - Static view of the program
- Object Diagrams
  - Show objects and their relationships at one moment in time during the execution of the program
  - Dynamic view of the program

Implementation: NumberDisplay

```
public class NumberDisplay
{
    private int limit;
    private int value;

   /* Constructor and
    * methods omitted
    */
}
```

Implementation: ClockDisplay

```
public class ClockDisplay
{
    private NumberDisplay hours;
    private NumberDisplay minutes;
    private NumberDisplay seconds;

    /* Constructor and
    * methods omitted
    */
}
```

- Importantly, however, we cannot describe all elements in a Java program using Class and Object diagrams:
- Java defines two very different categories of data type:
- primitive types
  - int, char, boolean etc.
- object types
  - ArrayList, LinkedList, NumberDisplay, ClockDisplay

- There are eight primitive data types supported by Java
  - byte (8-bit singed integer)
  - short (16-bit signed integer)
  - int (32-bit signed integer)
  - long (62-bit signed integer)
  - float (32-bit floating point number)
  - double (64-bit floating point number)
  - boolean (single bit true/false)
  - char (16-bit Unicode character)

- All primitive types are predefined by Java.
- Object types originate from classes.
  - Some of which are predefined and
  - Some of which we can write ourselves
- The primitive types are non-object types.
  - Have no constructors, accessors, mutators
- This is the reason why Java is not a completely object oriented language

```
//asignement of a primitive to a variable
//without a call to a constructor
int a = 6;

//asignment of an Object to a variable
//using a constructor call
MyObject obj = new MyObject(500);
```

- Primitive variables hold a value
- Object variables hold a reference to an Object

- There are two ways of passing arguments to methods in many programming languages including Java
- **Call-by-value**: A copy of the actual parameter is passed to the formal parameter of the called method. Any change made to the formal parameter will have no effect on the actual parameter.
- **Call-by-reference**: the caller gives the called method the ability to directly access to the callers data and to modify that data if the called method so chooses.

- It may be useful to think of primitives as passed by value
  - Since a copy is passed to each method that requires a primitive type parameter
- And objects as passed by reference
  - since an object passed to a method can be changed **permanently** within that method

- Returning to our DigitalClock, we will use the incremental development approach
- Starting with class comments and a class definition for the NumberDisplay class

```
/**
 * The NumberDisplay class represents a digital number
 * display that can hold values from zero to a given
 * limit. The limit can be specified when creating
 * the display. The values range from zero (inclusive)
 * to limit-1. If used, for example, for the seconds
 * on a digital clock, the limit would be 60,
 * resulting in display values from 0 to 59.
 * When incremented, the display automatically rolls
 * over to zero when reaching the limit.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2001.05.26
 */
public class NumberDisplay{
    // variables, constructors and methods go here
}
```

- followed by a definition of the fields needed by NumberDisplays

```
private int limit;
private int value;
```

- Next, we define constructor(s) for NumberDisplay

```
/**
 * Constructor for objects of class NumberDisplay
 */
public NumberDisplay(int rollOverLimit){
    limit = rollOverLimit;
    value = 0;
}
```

- Next, we can define the **accessors** for the NumberDisplay class
- ...and the **mutators**

```
    // Return the current value of the number display
    public int getValue(){
        return value;
    }

    /**
     * Set the value of the display to the new
     * specified value. If the new value is
     * less than zero or over the limit, do nothing.
     */
    public void setValue(int replacementValue){
        if((replacementValue >= 0) &&
           (replacementValue < limit)){
               value = replacementValue;
           }
    }
```

```
/**
 * Return the display value (that is, the current
 * value as a two-digit String. If the value is
 * less than ten, it will be padded with
 * a leading zero).
 */
public String getDisplayValue(){
    if(value < 10)
        return "0" + value;
    else
        return "" + value;
}
```

```
/**
 * Increment the display value by one, rolling
 * over to zero if the limit is reached.
 */
public void increment()
{
    value = (value + 1) % limit;
}
```

- Having written code that defines the NumberDisplay class
- i.e. a template for all NumberDisplay Objects
- we can now write a ClockDisplay class
- That uses three NumberDisplay Objects

```
/**
 * The ClockDisplay class implements a digital clock
 * display for a European-style 24 hour clock.
 * The clock shows hours and minutes. The range of
 * the clock is 00:00:00 (midnight) to 23:59:59
 * (one second before midnight).
 *
 * The clock display receives "ticks" (via the
 * timeTick method) every second
 * and reacts by incrementing the display. This is
 * done in the usual clock fashion:
 * the hour increments when the minutes roll over
 * to zero and minutes increment when
 * seconds roll over to zero.
 *
 * @author Michael Kolling and David J. Barnes
 * @version 2001.05.26
 */
```

```
public class ClockDisplay{

    private NumberDisplay hours;
    private NumberDisplay minutes;
    private NumberDisplay seconds;
    private String displayString;

}
```

```
/**
     * Constructor for ClockDisplay objects.
     * This constructor
     * creates a new clock set at 00:00:00.
     */
    public ClockDisplay(){
        hours = new NumberDisplay(24);
        minutes = new NumberDisplay(60);
        seconds = new NumberDisplay(60);
        updateDisplay();
    }
```

```
/**
 * Constructor for ClockDisplay objects.
 * This constructor creates a new clock
 *  set at the time specified by the
 * parameters.
 */
public ClockDisplay(int hour, int minute, int
    second){
     hours = new NumberDisplay(24);
     minutes = new NumberDisplay(60);
     seconds = new NumberDisplay(60);
     setTime(hour, minute,second);
}
```

```
    /**
     * This method should get called once every
     * second - it makes the clock
     * display go one minute forward.
     */
    public void timeTick(){
        //some code
    }

    /**
     * Set the time of the display to the specified
     * hour and minute and second.
     */
    public void setTime(int hour, int minute, int
        second){
        //some code
    }
```

```
 /**
  * Return the current time of this display
  * in the format HH:MM:SS.
  */
public String getTime(){
    //some code
}

/**
  * Update the internal string that
  * represents the display.
  */
private void updateDisplay(){
    //some code
}
```

- Note: The ClockDisplay class creates and uses three NumberDisplay Objects
  - new NumberDisplay(parameter-list)
    - ★ creates a new NumberDisplay Object
    - ★ i.e. we execute the constructor of that class
    - ★ this involves creating sufficient memory to store the values of primitive instance variables and references to object instance variables.

- Public methods:
  - public void increment() can be called externally
- Private methods
  - private void updateDisplay()
  - can only be called internally used for auxiliary methods

Aside : this keyword

```
public class Point {
    public int x = 0;
    public int y = 0;

    //constructor
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

- this is used to go out of the scope of the constructor to class level
  - this always refers to the current object.
  - can also used for methods
  - for internal methods calls and access to instance fields Java automatically inserts this:
  - updateDisplay $->$ this.updateDisplay

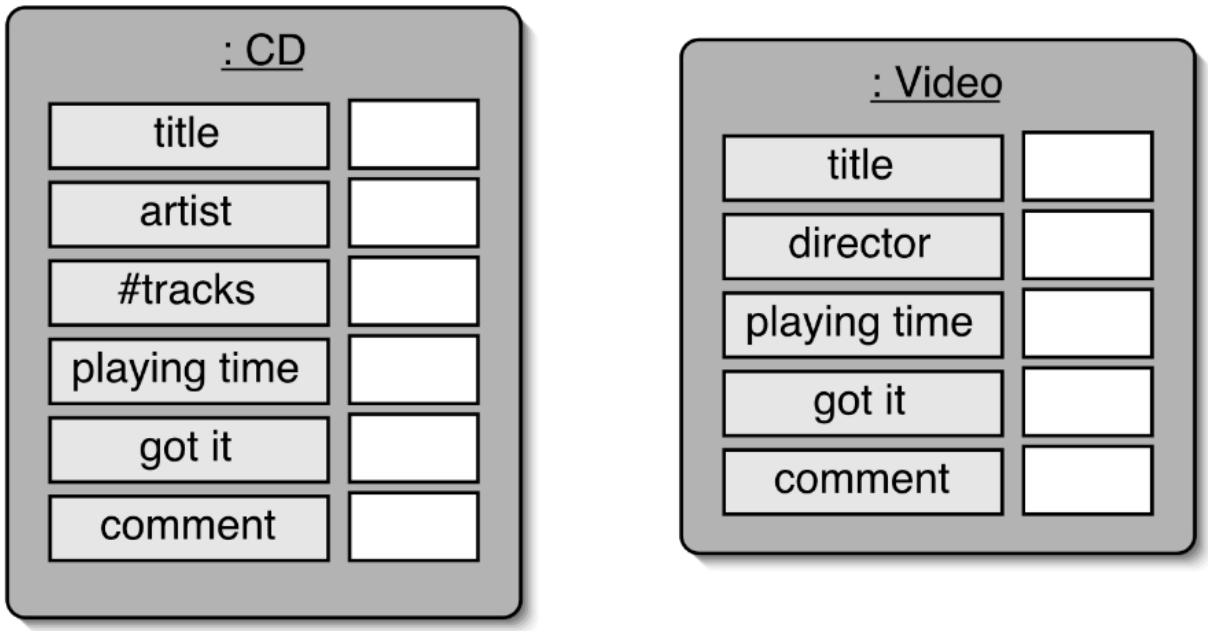
Summary

- Recapped on and explored further Classes and Objects
- Variables
- Constructors
- Accessor and Mutators
- Methods

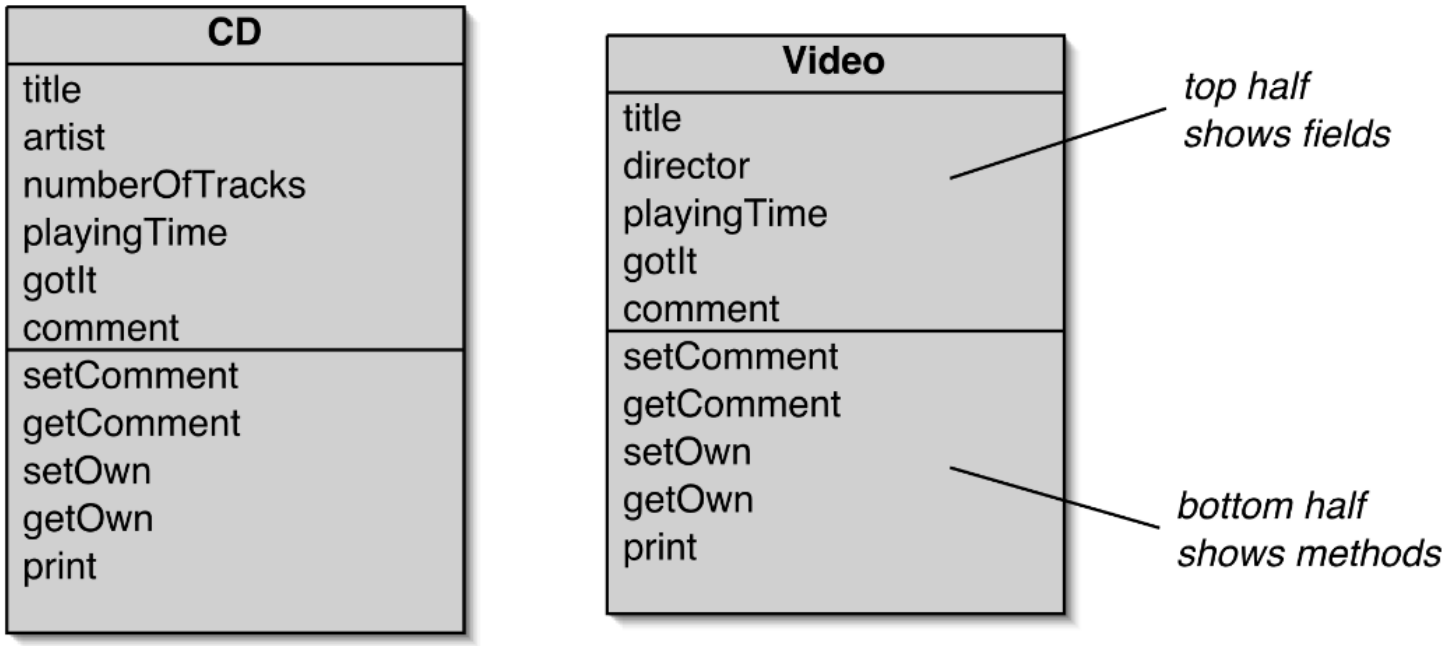
Inheritance

- **Database of Multimedia Entertainment**
- Stores details about CDs and videos
  - CD: title, artist, # tracks, playing time, got-it, comment
  - Video: title, director, playing time, got-it, comment
- Allows (later) to search for information or print lists

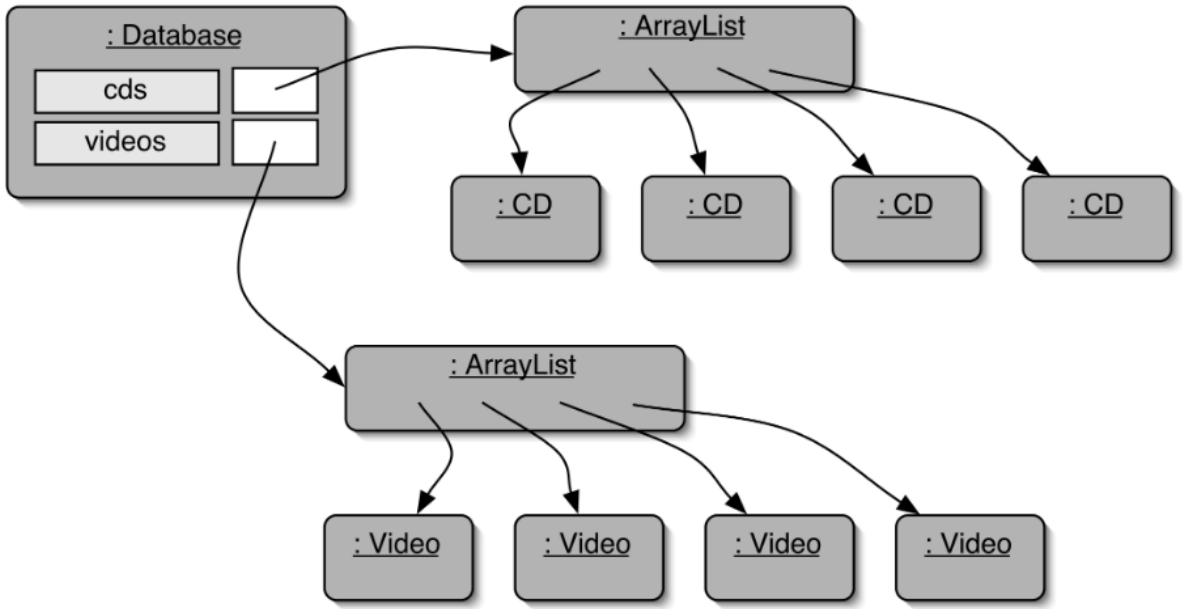
DOME Object Diagram



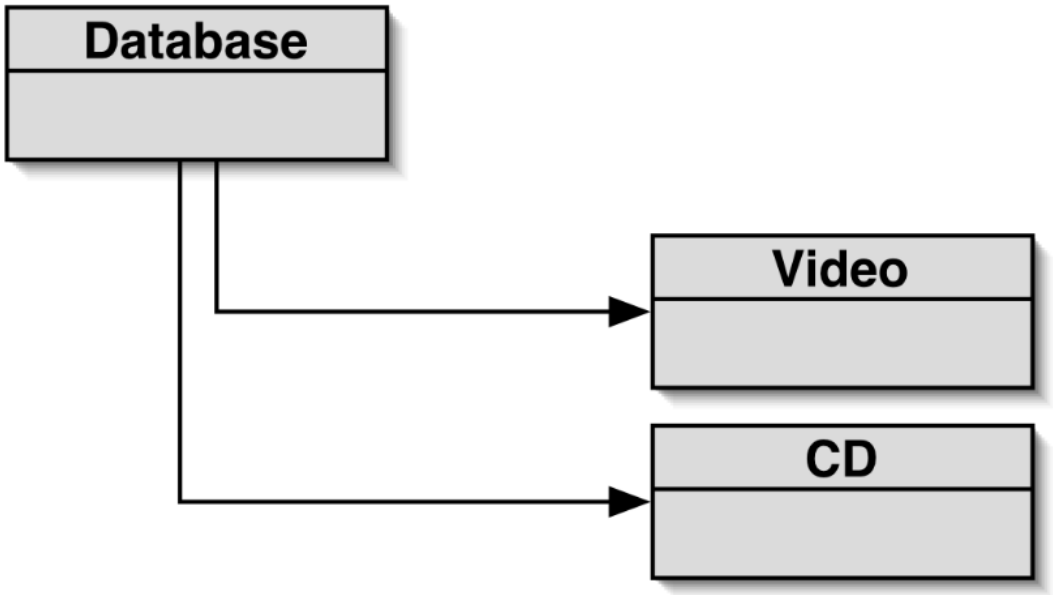
DOME Class Diagram



DOME Object Diagram continued...



DOME Class Diagram continued...



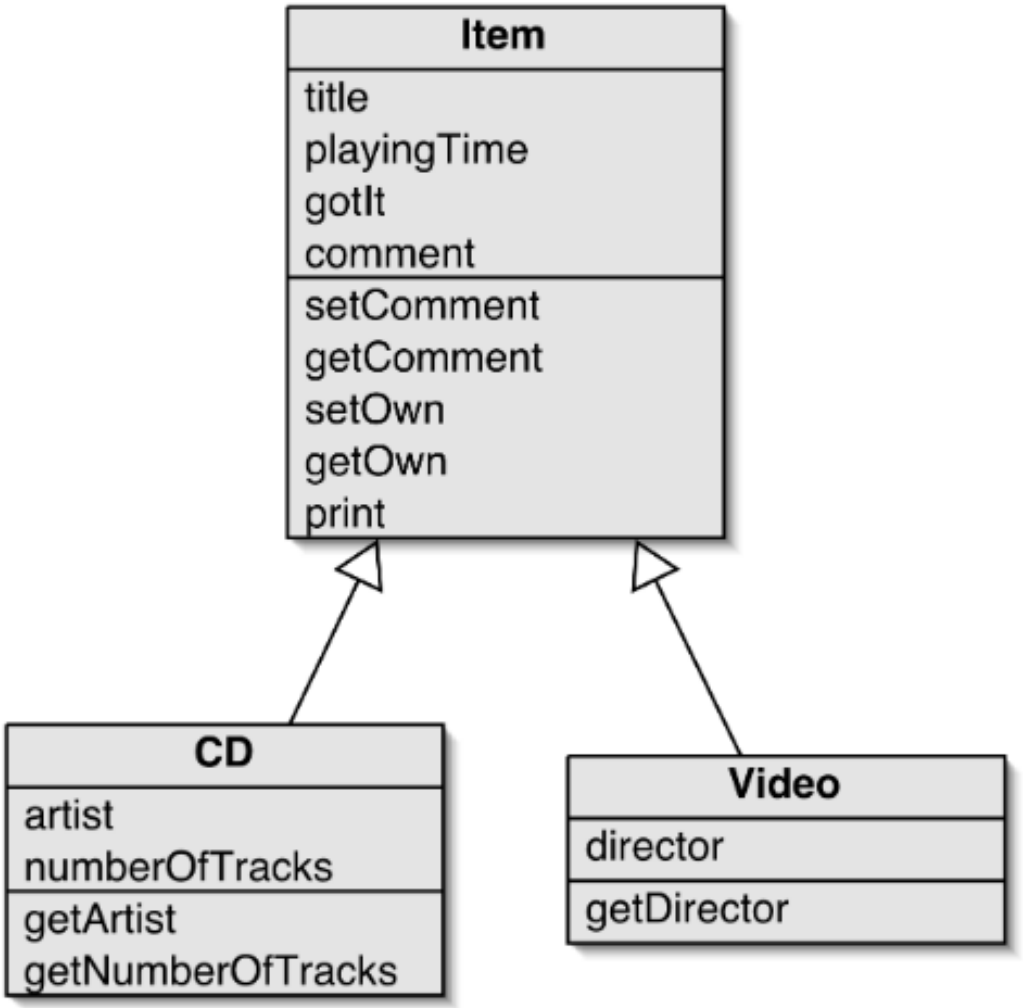
```
public class CD {
  private String title ;
  private String artist ;
  private String comment;

  public CD( String theTitle , String theArtist ) {
  title = theTitle;
  artist = theArtist;
  comment = "";
  }

  public void setComment (String newComment)  {...}

  public String getComment()  {...}

  public void print ()  {...}

}
```

```
public class Video{
  private String title ;
  private String director ;
  private String comment;

  public Video( String theTitle, String theDirect ){
    title = theTitle;
    director=theDirect;
    comment="";
  }

  public void setComment(String newComment) { ... }

  public String getComment() { ... }

  public void print()  { ... }

}
```

```
public class Database{
   private ArrayList cds;
   private ArrayList videos;

   public void list(){

      for(i=0, i<cds.size(); i++){
         cd.get(i).print();
         System.out.println();
      }

      for(i=0, i<videos.size(); i++){
         videos.get(i).print();
         System.out.println();
      }
   }
}
```

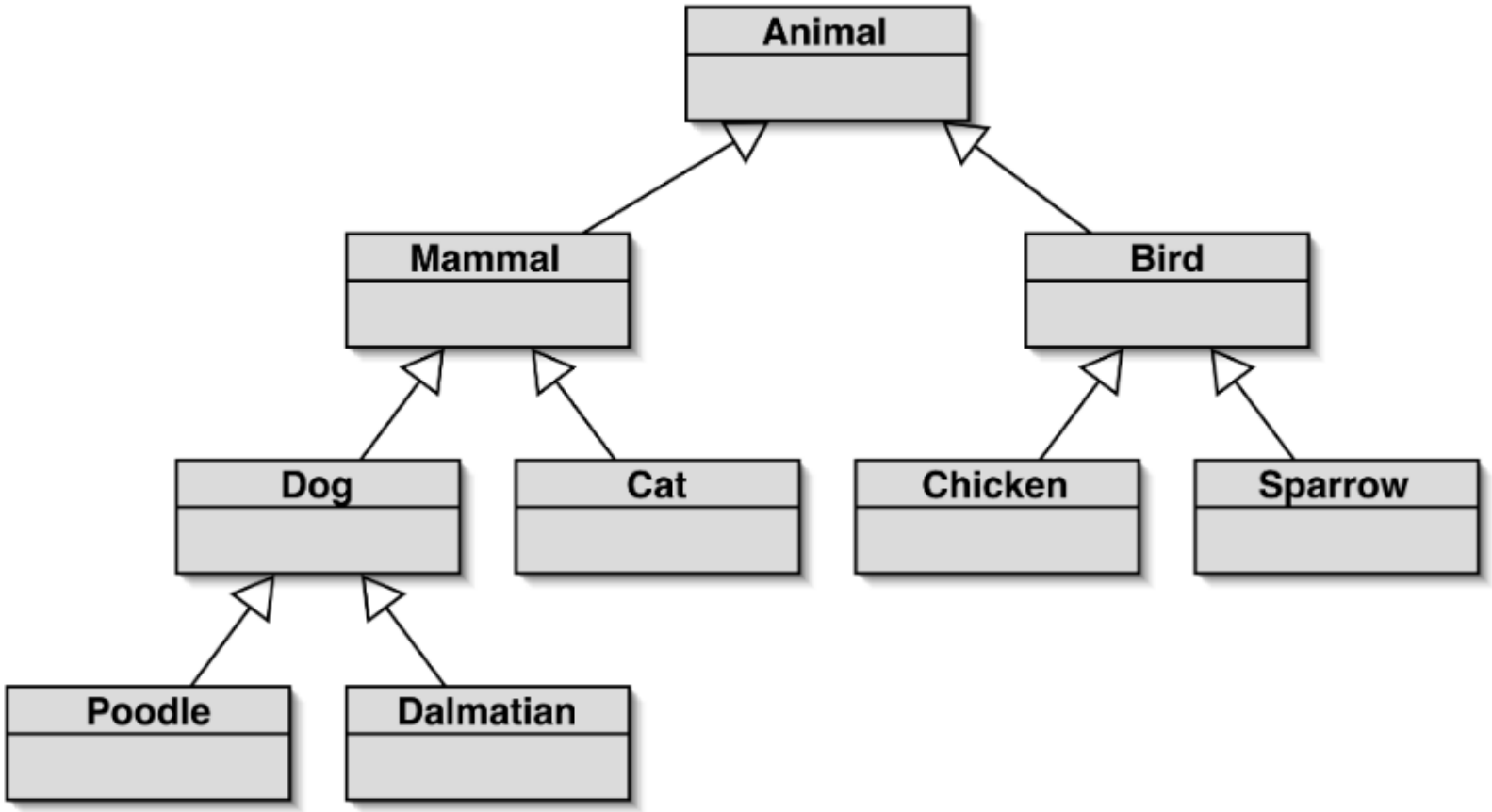
- Critique of DOME
- Code duplication
  - CD and Video classes very similar (large part are identical)
  - makes maintenance difficult/more work
  - introduces danger of bugs through incorrect maintenance
- Code duplication also in Database class

DOME Object Diagram Using Inheritance



- Using Inheritance
  - define one superclass : Item
  - define subclasses for Video and CD
  - the superclass defines common attributes
  - the subclasses inherit the superclass attributes
  - the subclasses add own attributes

DOME Object Diagram Using Inheritance



```
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

}
```

```
public class Video extends Item{

  private String director;

  //constructors and methods omitted

}
```

```
public class Item{

   private String Title;
   private int playingTime;
   private boolean gotIt;
   private String comment;

   public Item(Sring theTitle, int time){
      title = theTitle;
      playingTime = time;
      gotIt = false;
      comment = "";
   }

   //methods omitted

}
```

```
public class CD extends Item{
  private String artist;
  private int numberOfTracks;

  public CD(String theTitle, String theArtist,
                  int tracks, int time){
    super(theTitle, time);
    artitst = theArtist;
    numberOfTracks = tracks;
  }

  //methods omitted

}
```

- Subclass constructors must always contain a 'super' call.
  - If none is written, the compiler inserts one (without parameters)
  - Works only, if the superclass has a constructor without parameters
  - Must be the first statement in the subclass constructor.

- Why do we care?

- Inheritance (so far) helps with:
  - Avoiding code duplication
  - Code reuse
  - Easier maintenance
  - Extendibility

```
public class Database {
  private ArrayList<Item> items ;

  // Construct an empty Database
  public Database( ) {
    items = new ArrayList<Item>() ;
  }

  // Add an item to the database
  public void addItem ( Item theItem ) {
    items.add(theItem);
  }
  ...
}
```

http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

```
/**
* Print a list of all currently stored CDs and
* videos to the text terminal .
**/

public void list () {
   for(i=0; i<items.size; i++){
      Item item = items.get(i);
      item.print();
      System.out.println() ;
   }
}
```

- Subtyping

- First, we had:
  - public void addCD(CD theCD)
  - public void addVideo(Video theVideo)
- Now, we have:
  - public void addItem(Item theItem)
  - We call this method with:

```
Video myVideo = new Video(...);
database.addItem(myVideo);
```

- Subclasses and subtyping

- Classes define types.
- Subclasses define subtypes.
- Objects of subclasses can be used where objects of supertypes are required.
- This is called substitution.

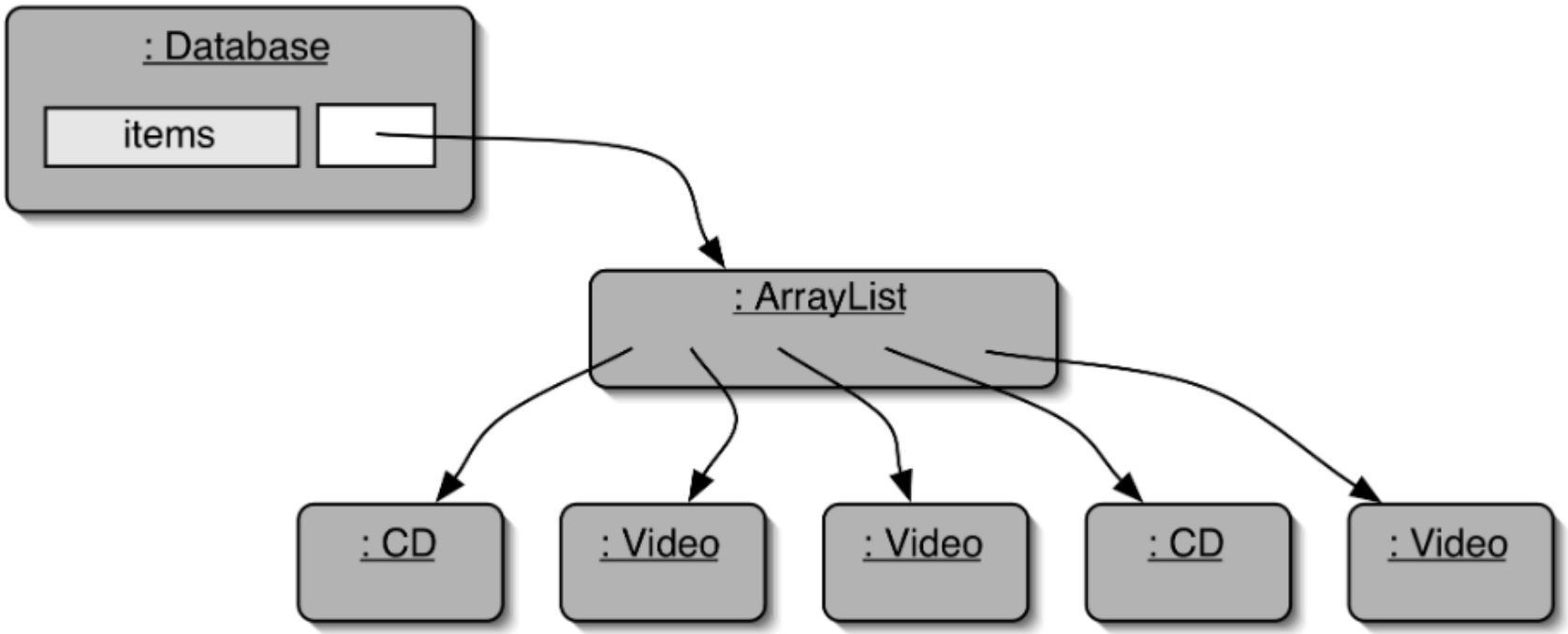
- Subclass objects may be assigned to superclass variables
- e.g. Car extends Vehicle and Bicycle extends Vehicle

```
Vehicle v1 = new Vehicle();
Vehicle v2 = new Car();
Vehicle v3 = new Bicycle();
```

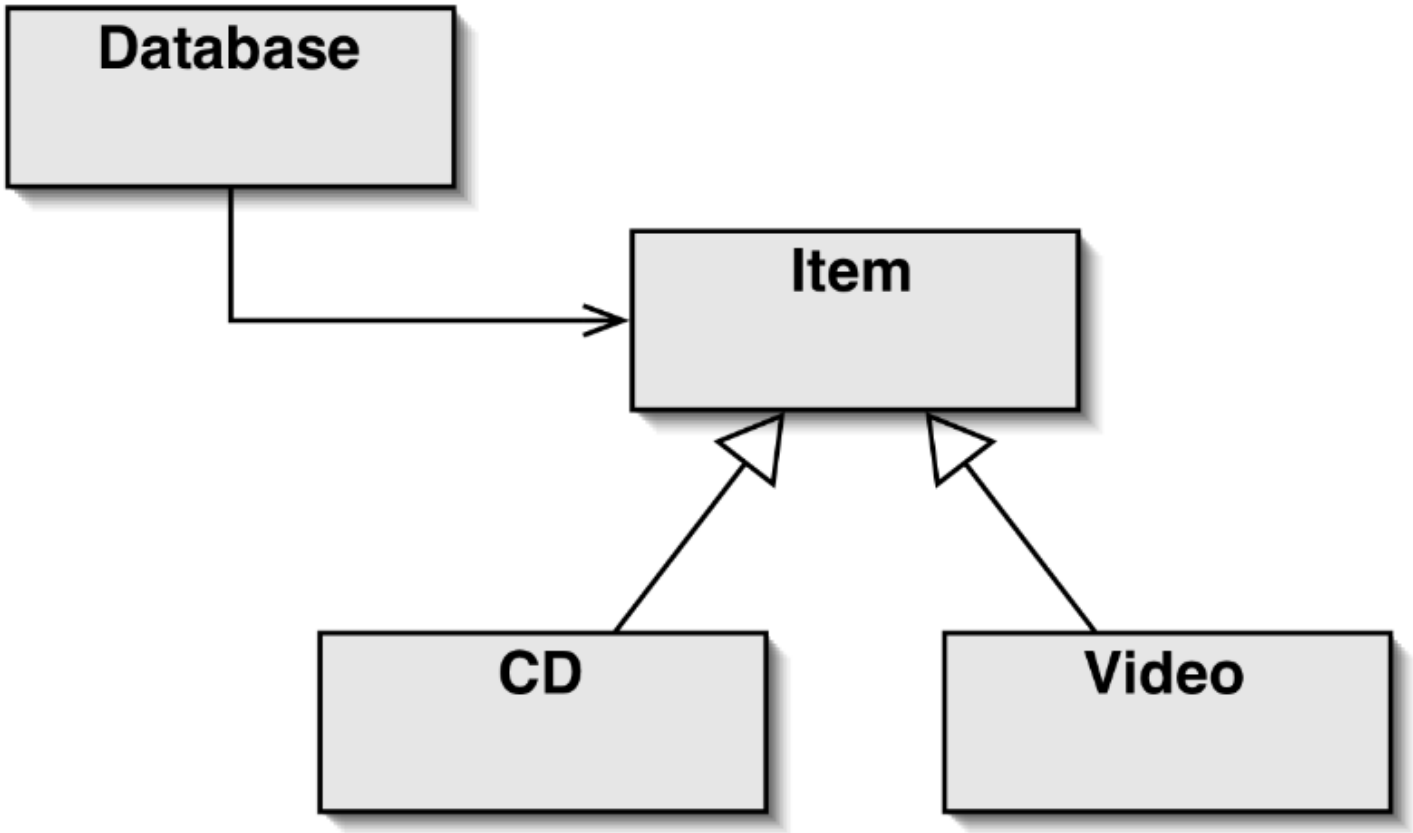
- Subclass objects may be passed to superclass parameters

```
public class Database{
  public void addItem (Item theItem){
    ....
  }
}

//code in another method
Video video = new Video(...);
CD cd = new CD(...);

database.addItem (video);
database.addItem (cd);
```

Object Diagram Illustrating Inheritance



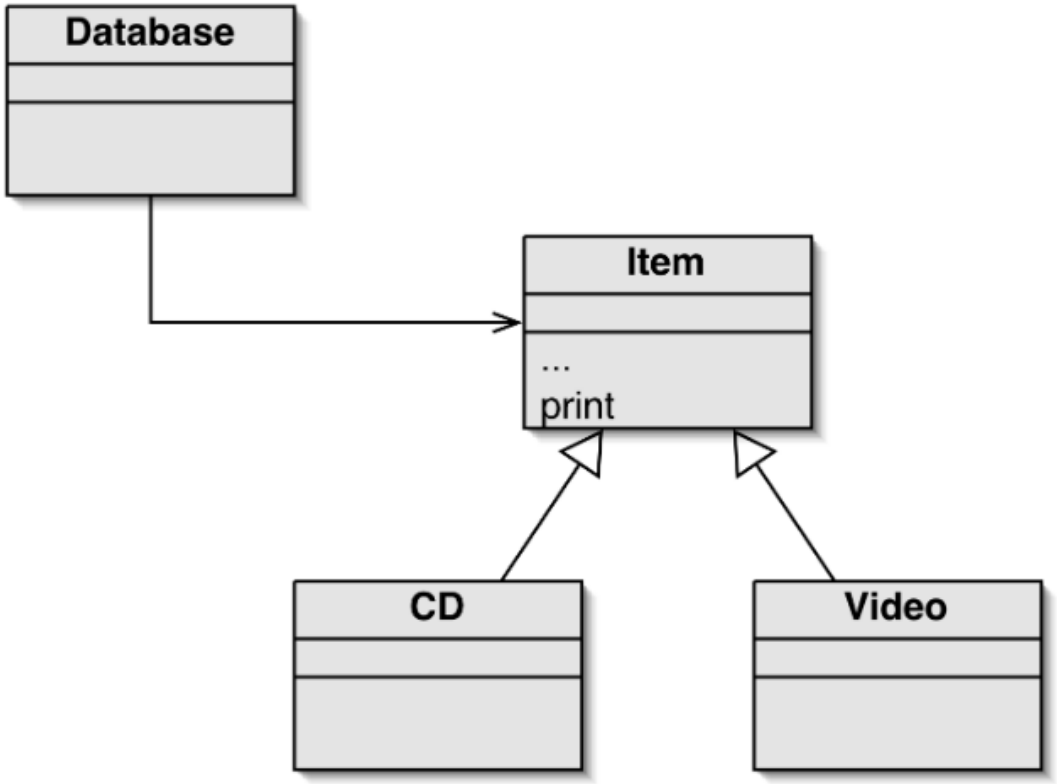
Class Diagram Illustrating Inheritance



- Review

- Inheritance allows the definition of classes as extensions of other classes.
- Inheritance
  - avoids code duplication
  - allows code reuse
  - simplifies the code
  - simplifies maintenance and extending
- Variables can hold subtype objects.
- Subtypes can be used wherever supertype objects are expected (substitution).

- Polymorphic variables
- Object variables in Java are polymorphic.
  - They can hold objects of more than one type.
- They can hold objects of the declared type, or of subtypes of the declared type.

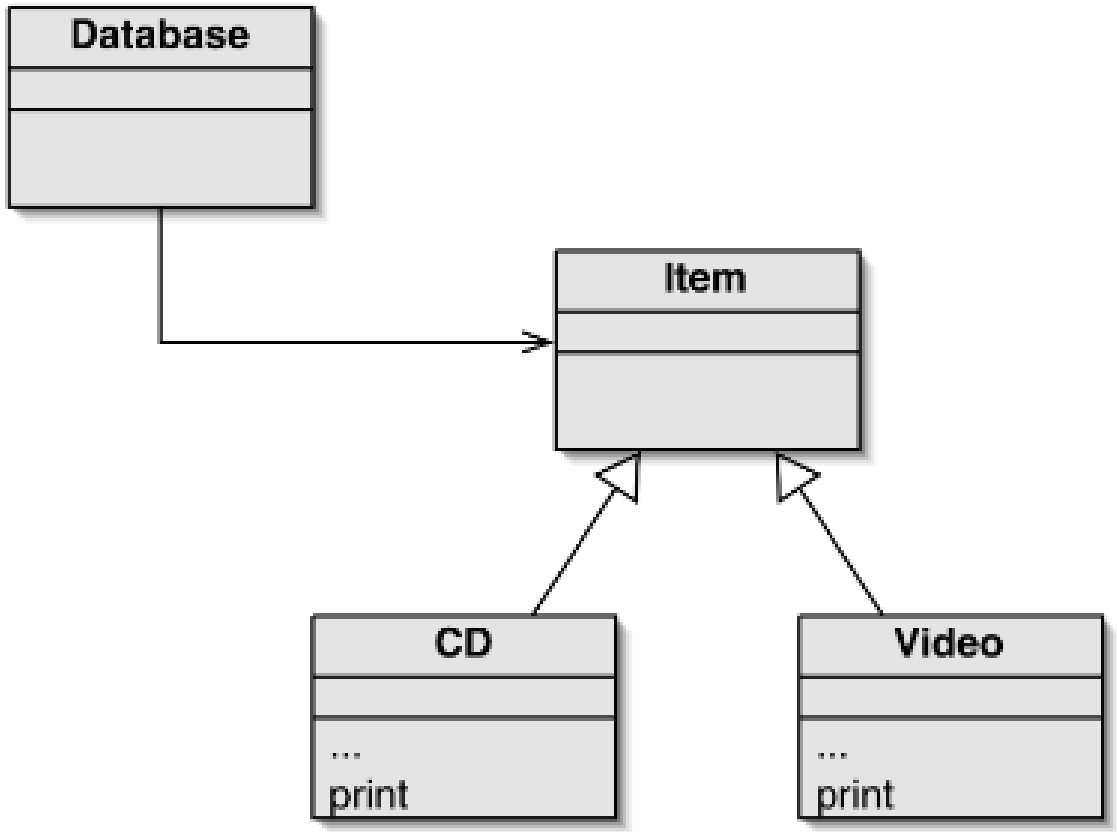

Database
Item
...
print
CD
Video

- The print method in Item only prints the common fields.
- Inheritance is a one-way street:

- A subclass inherits the superclass fields.
- The superclass knows nothing about its subclasss fields.

- Attempting to Solve the Problem.

- Place print where it has access to the information it needs.



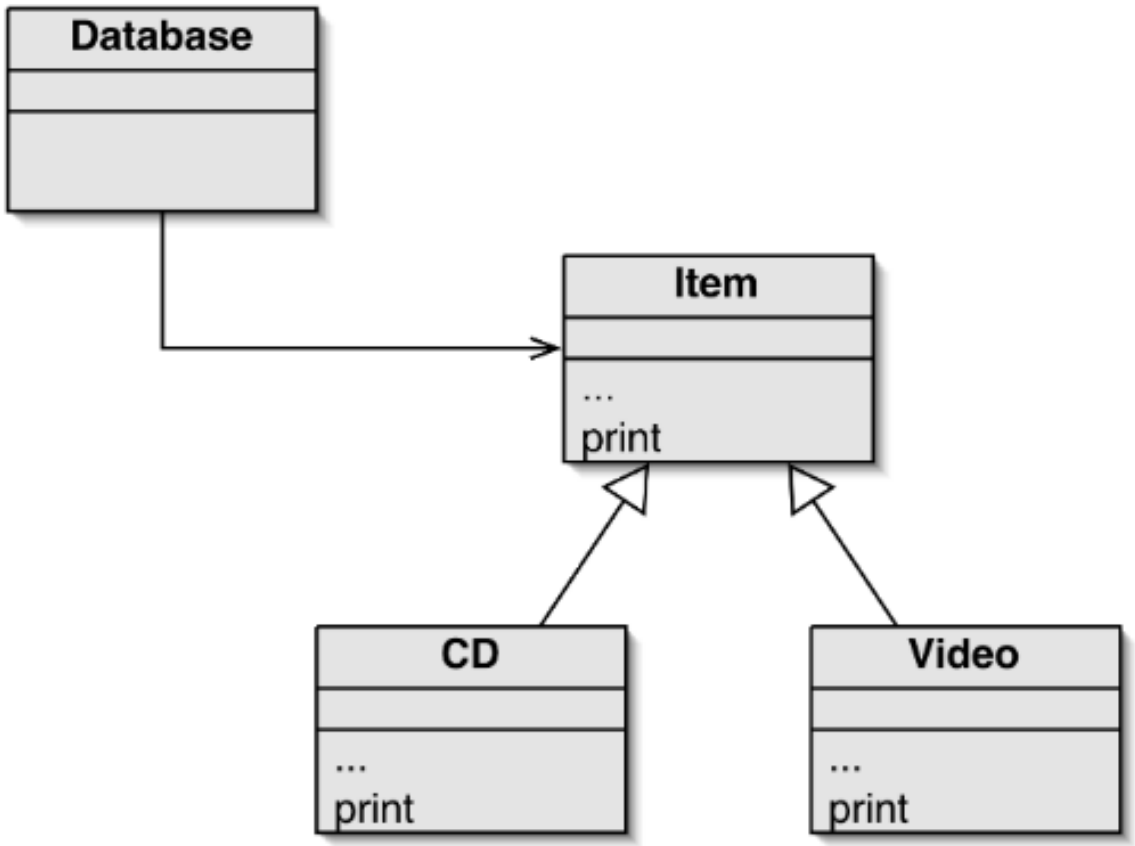
- Each subclass has its own version.
- But Items fields are private.
- Database cannot find a print method in Item.

- To solve our problem we need to introduce...
- some new terminology:
  - static type
  - dynamic type
  - method dispatch/lookup

- Static Type and Dynamic Type
- The declared type of a variable is its static type.
- The type of the object a variable refers to is its dynamic type.
- The compilers job is to check for static-type violations.

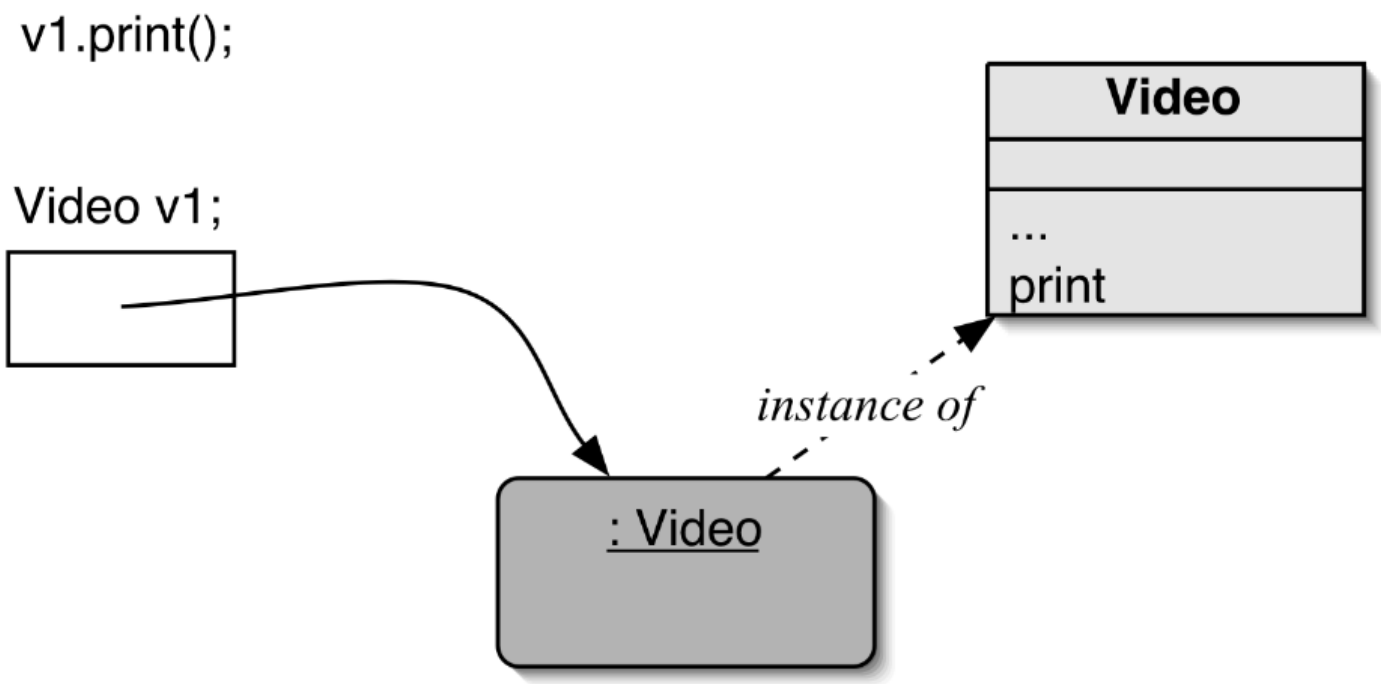
```
class Alpha{}
class Beta extends Alpha{}
class Fruit extends Beta{}

Fruit f = new Fruit(); //static=Fruit, dynamic=Fruit
Alpha a = f; //static=Alpha, dynamic=Fruit

Fruit f = a //static type violation
```

- Returning to our problem...
- The Solution: Overriding
  - print method in both super- and subclasses
  - Satisfies both static and dynamic type checking

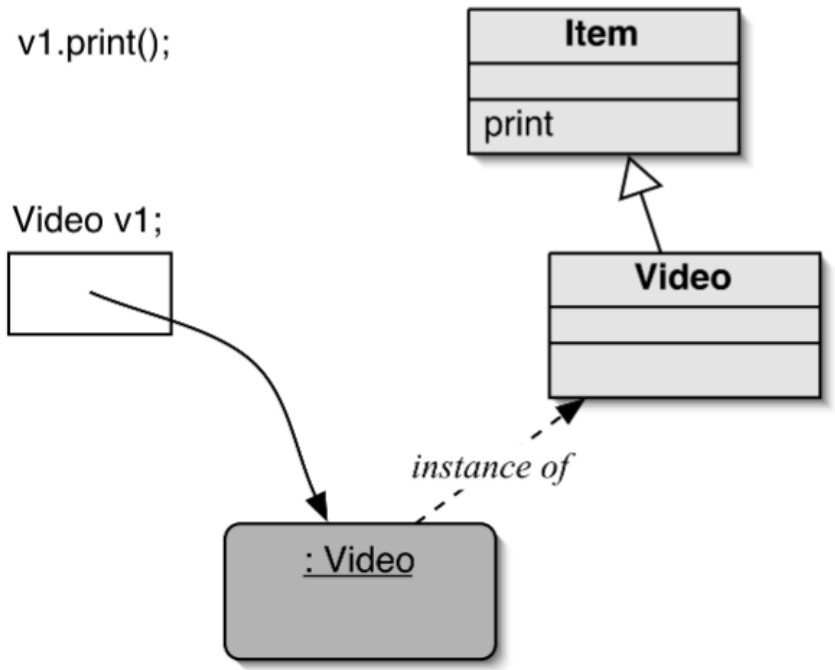

Database
Item
...
print
CD
...
print
Video
...
print

- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass satisfies static type check.
- Subclass method is called at runtime it overrides the superclass version.
- What becomes of the superclass version?

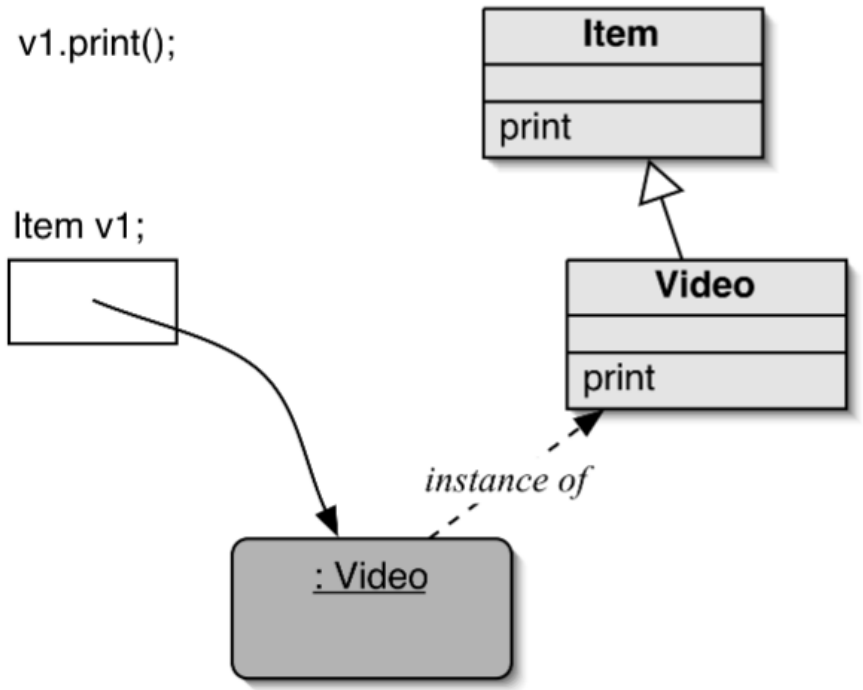
- Method Lookup 1
  - No inheritance or polymorphism.
  - The obvious method is selected.



- Method Lookup 2
  - Inheritance but no overriding
  - The inheritance hierarchy is ascended, searching for a match.



- Method Lookup 3
  - Polymorphism and overriding.
  - The first version found is used.



- Method Lookup Summary

- The variable is accessed.
- The object stored in the variable is found.
- The class of the object is found.
- The class is searched for a method match.
- If no match is found, the superclass is searched.
- This is repeated until a match is found, or the class hierarchy is exhausted.
- Overriding methods take precedence.

- Super call in methods
- Overridden methods are hidden ...
- ... but we often still want to be able to call them.
- An overridden method can be called from the method that overrides it
  - super.method(...)
  - Compare with the use of super in constructors.

```
public class CD{

   ...

   public void print (){
      super.print();
      System.out.println (""+artist);
      System.out.println("tracks:" + numberofTracks);
   }
}
```

- We have been discussing polymorphic method dispatch.
- A polymorphic variable can store objects of varying types.
- Method calls are polymorphic.
  - The actual method called depends on the dynamic object type.

- Methods in Object are inherited by all classes.
- Any of these may be overridden.
- The toString method is commonly overridden:
- public String toString()
  - Returns a string representation of the object.

```
public class Item{

  ...

   public String toString (){
     String line1=title + " : " + playingTime + " mins"
         );
     if(gotIt) {
       return line1 + "\n" + comment + "\n");
     }
     else {
       return line1 + "\n" + comment + " need to buy" +
           "\n");
     }
   }
}
```

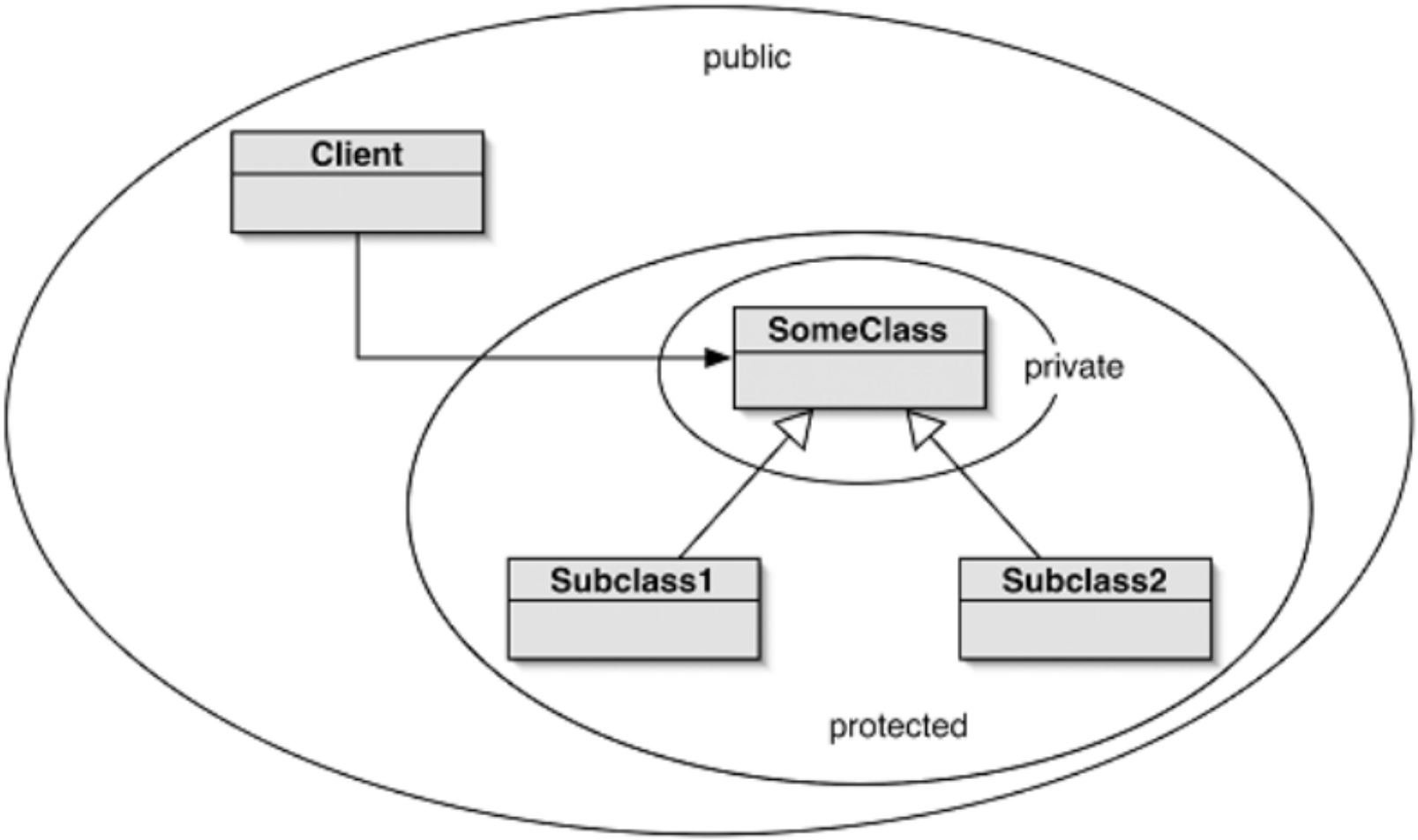
- Explicit print methods can often be omitted from a class:

```
System.out.println(item.toString()) ;
```

- Calls to println with just an object automatically result in
- toString being called:

```
System.out.println(item);
```

- Private access in the superclass may be too restrictive for a subclass.
- The closer inheritance relationship is supported by protected access.
- Protected access is more restricted than public access.
- We still recommend keeping fields private.
- Define protected accessors and mutators.

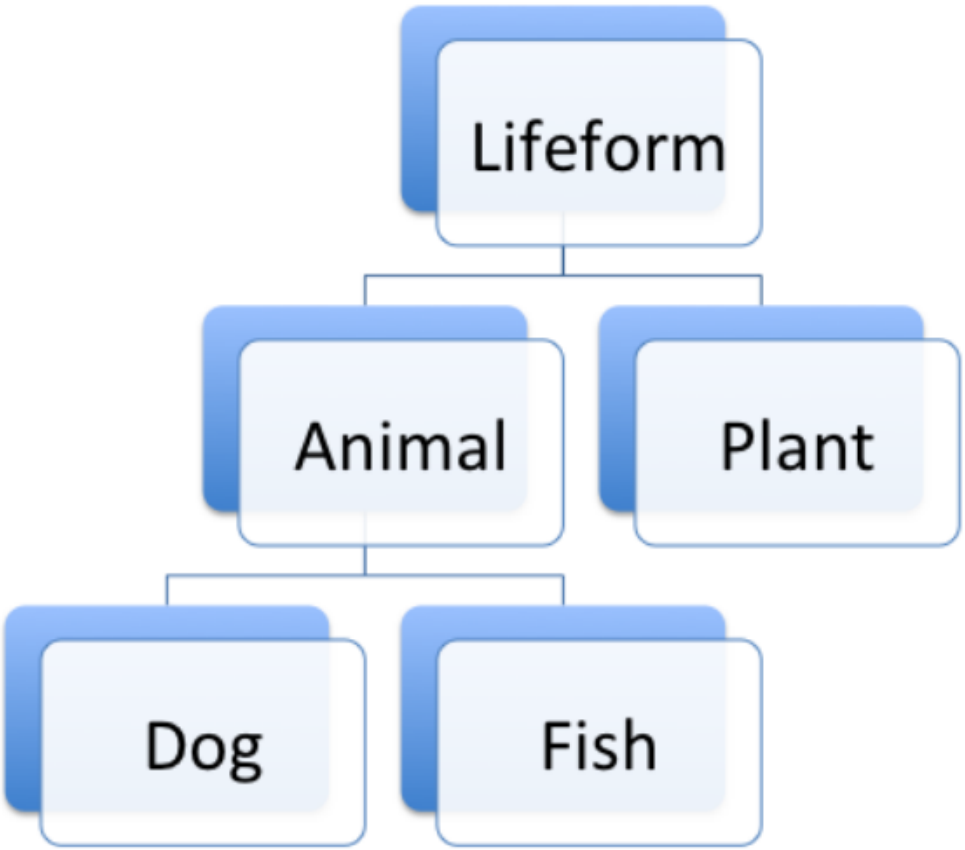

public
Client
SomeClass
private
Subclass1
Subclass2
protected

- Review

- The declared type of a variable is its static type.
- Compilers check static types.
- The type of an object is its dynamic type.
- Dynamic types are used at runtime.
- Methods may be overridden in a subclass.
- Method lookup starts with the dynamic type.
- Protected access supports inheritance.

Abstract Classes and Interfaces

- We have looked at the concept of overridding methods (extending or rewriting methods in subclasses when they already exist in superclasses)
- In some situations, however, we may want to include methods in superclasses but only allow them to be used by subclasses


Lifeform
Animal
Plant
Dog
Fish

- For example, we may 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 methods and classes
- The makeSound() method of Animal

```
abstract void makeSound();
```

- Abstract classes and methods
  - 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.

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

    public int getAverageAge(){
        return averageAge;
    }

    // more methods

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

- Note, however, that abstract classes are still classes
  - more specifically, partially implemented classes
- So the rules that govern implemented classes also govern abstract classes
  - No subclass can have two (or more) abstract parents

- Note also 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

- Finally (for now) note that abstract superclasses can have abstract subclasses
- If, for some reason we don't want to implement all the methods in an abstract superclass (parent), we can
  - implement some methods
  - leave other methods as abstract
  - and declare the subclass (child) as abstract
- This approach will leave the subclass's subclasses to instantiate the other abstract methods (or their subclasses, subclass's subclasses etc.)

- At some point, however, we may want an even looser relationship between superclass and subclass
- For example, we may want to specify a superclass in which no methods are implemented
- Leaving subclasses to do all of the implementation
  - e.g we might want to specify an interface which describes the methods provided by all classes that can be drawn to the screen (the Drawable interface)
  - we would not, however specify anything about the way that drawing would occur at the level of the interface
  - all the implementation would be described elsewhere

- In Java, this "specify but implement nothing" approach is achieved through the use of Interfaces
- The fact that a subclass implements a particular interface is (rather unsurprisingly) implemented through the use of the **implements** keyword

- Interfaces
  - 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
  - uses interface instead of class
  - all methods are public (no need to mention)
  - all methods are abstract (no need to mention)
  - no constructors
  - only constant fields are allowed (public static final)

- Animal interface

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

    String makeSound();

}
```

- Classes implement an interface

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

- Abstract classes can also implement interfaces

```
abstract class Canine implements Animal {
        ...
 }
```

- Interfaces are not classes (though they cannot share a name with a class)
- They 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
- 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

- Unlike abstract classes interfaces cannot contain implemented methods (methods for which the body code has been written)
- Interfaces can only contain
  - method stubs
  - constants
- Interfaces can, however, extend other interfaces

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

```
Animal fox = new Fox();
```

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

Multiple Inheritance

- Having a class inherit directly from multiple ancestors.
- Each object oriented programming language has its own rules.
- How to resolve competing definitions?
  - Java forbids it for classes.
  - Java permits it for interfaces.
- No competing implementation.

Review

- Inheritance can provide shared implementation.
  - Concrete and abstract classes.
- Inheritance provides shared type information.
  - Interfaces.

- Abstract classes function as incomplete superclasses.
  - No instances.
- Abstract classes support polymorphism
- Abstract methods allow static type checking without requiring implementation

- Interfaces
  - Interfaces provide specification without implementation.
  - Interfaces are fully abstract.
  - Interfaces support polymorphism.
  - Java interfaces support multiple inheritance.