More about inheritance

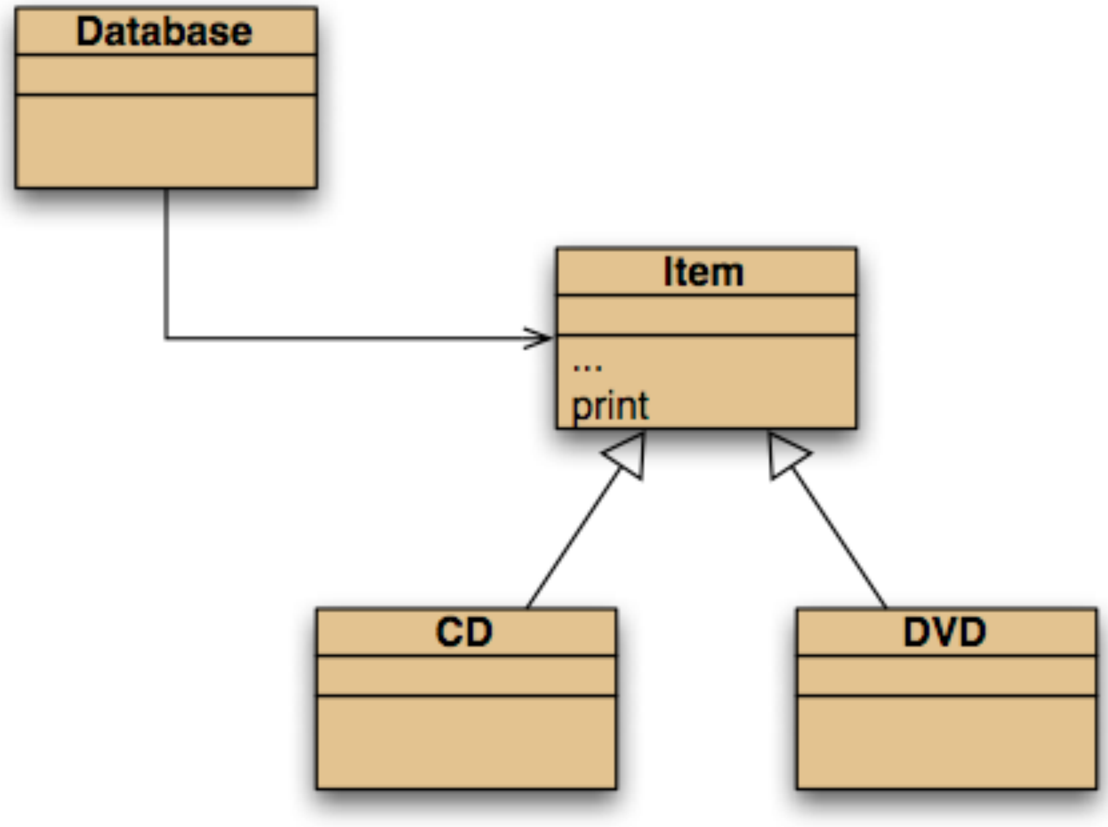
Exploring polymorphism

COMP1400
Week 11

Main concepts to be covered

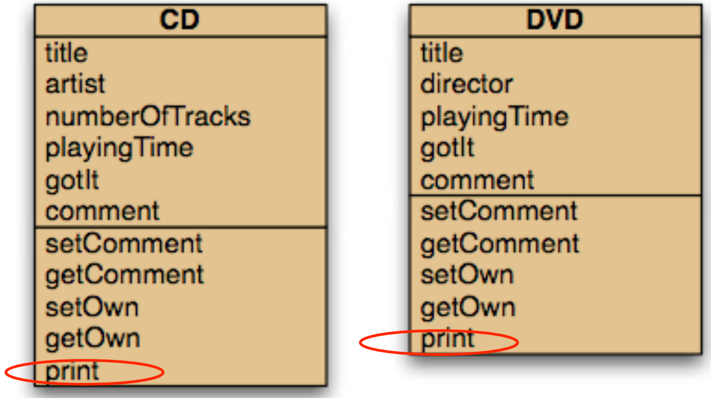
- method polymorphism
- static and dynamic type
- overriding
- dynamic method lookup
- protected access

The inheritance hierarchy



DoME classes

Without inheritance



CD
title
artist
numberOfTracks
playingTime
gotIt
comment
setComment
getComment
setOwn
getOwn
print

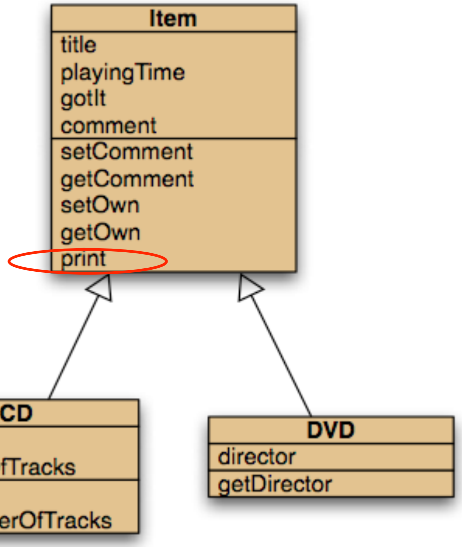
DVD
title
director
playingTime
gotIt
comment
setComment
getComment
setOwn
getOwn
print

```
class Database {


   private ArrayList<CD> cds;
   private ArrayList<DVD> dvds;
   ...
   public void list()
   {
       for(CD cd : cds)
       {
           cd.print();
           System.out.println();  // empty line between items
       }

       for(DVD dvd : dvds)
       {
           dvd.print();
           System.out.println();  // empty line between items
       }
   }
}
```

With inheritance



Item
title
playingTime
gotIt
comment
setComment
getComment
setOwn
getOwn
print

CD
artist
numberOfTracks
getArtist
getNumberOfTracks

DVD
director
getDirector

```
/**
 * Print a list of all currently stored CDs and
 * DVDs to the text terminal.
 */
public void list()
{
    for (Item item: items)
    {
        item.print();
        // Print an empty line between items
        System.out.println();
    }
}
```

Conflicting output

What we want

```
CD: A Swingin' Affair (64 mins)*
    Frank Sinatra
    tracks: 16
    my favourite Sinatra album

DVD: O Brother, Where Art Thou? (106 mins)
     Joel & Ethan Coen
     The Coen brothers’ best movie!
```

What we have

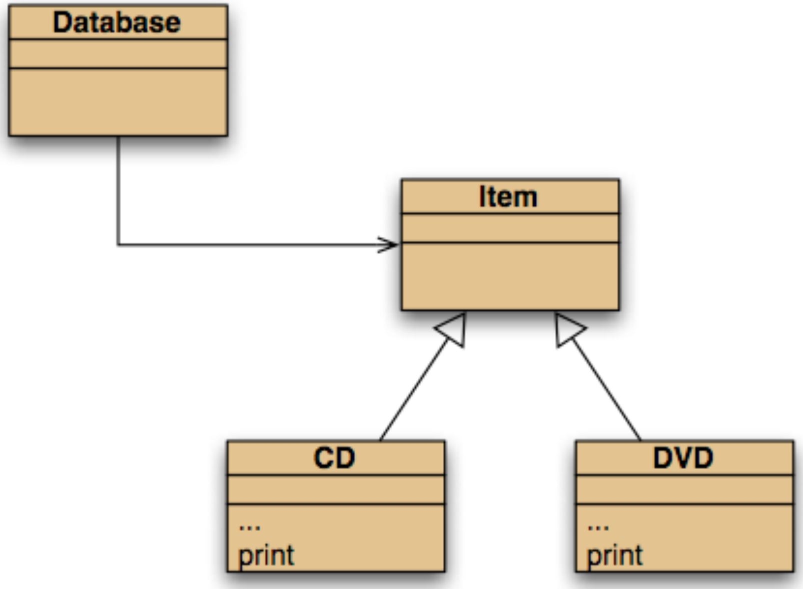
```
title: A Swingin' Affair (64 mins)*
       my favourite Sinatra album

title: O Brother, Where Art Thou? (106 mins)
       The Coen brothers’ best movie!
```

The problem

- 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 subclass's fields.

Attempting to solve the problem



- Place **`print`** where it has access to the information it needs.
- Each subclass has its own version.
- But **`Item`**’s fields are private.
- **`Database`** cannot find a **`print`** method in **`Item`**.

Static type and dynamic type

- A more complex type hierarchy requires further concepts to describe it.
- Some new terminology:
  - static type
  - dynamic type
  - method dispatch/lookup

Static and dynamic type

What is the type of c1?

```
Car c1 = new Car();
```

What is the type of v1?

The type of the variable `v1` is `Vehicle`

```
Vehicle v1 = new Car();
```

the type of the object stored in `v1` is `Car`.

Static and dynamic type

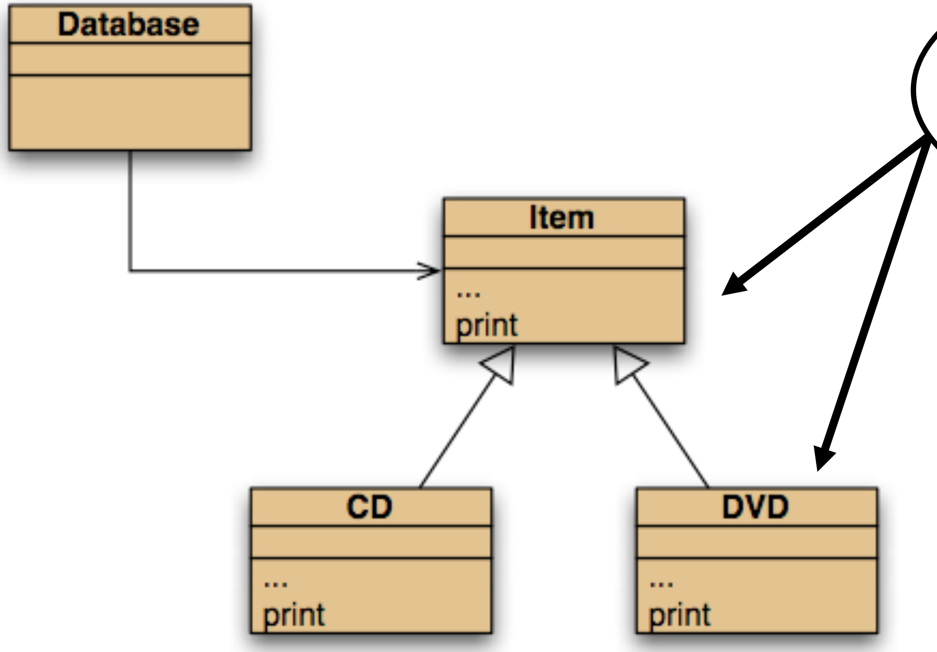
- The declared type of a variable is its *static type*.
- The type of the object a variable refers to is its *dynamic type*.
- The compiler's job is to check for static-type violations.

```
for (Item item: items)
{
    item.print();        // Compile-time error.
}
```

Static and dynamic type

- The **static type** of a variable v is the type as declared in the source code in the variable declaration statement.
- The **dynamic type** of a variable **v** is the type of the object that is currently stored in **v**.

Overriding: the solution



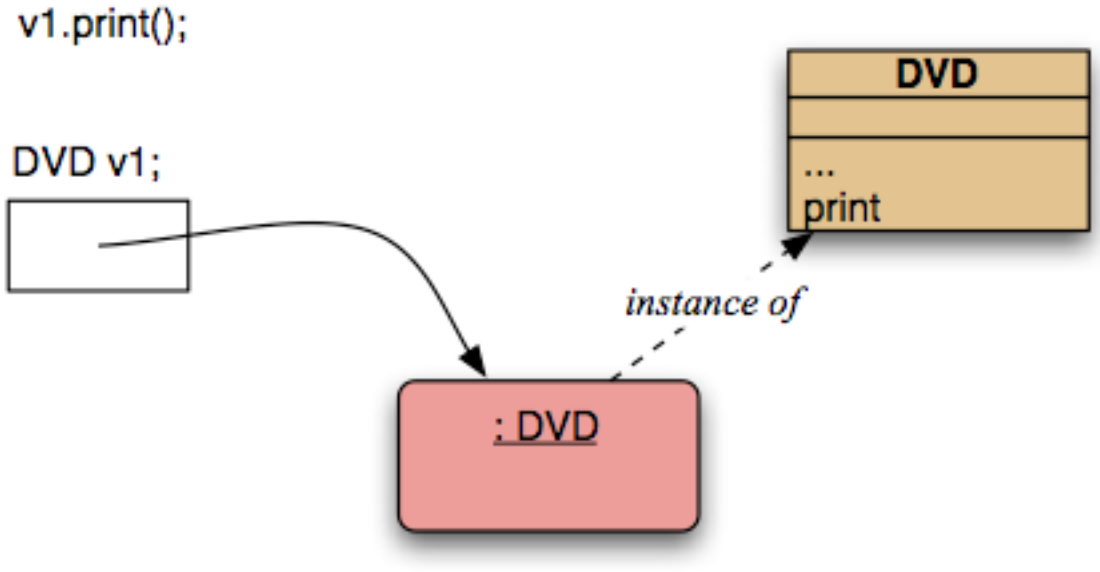
`print` method in both super- and subclasses.

Satisfies both static and dynamic type checking.

Overriding

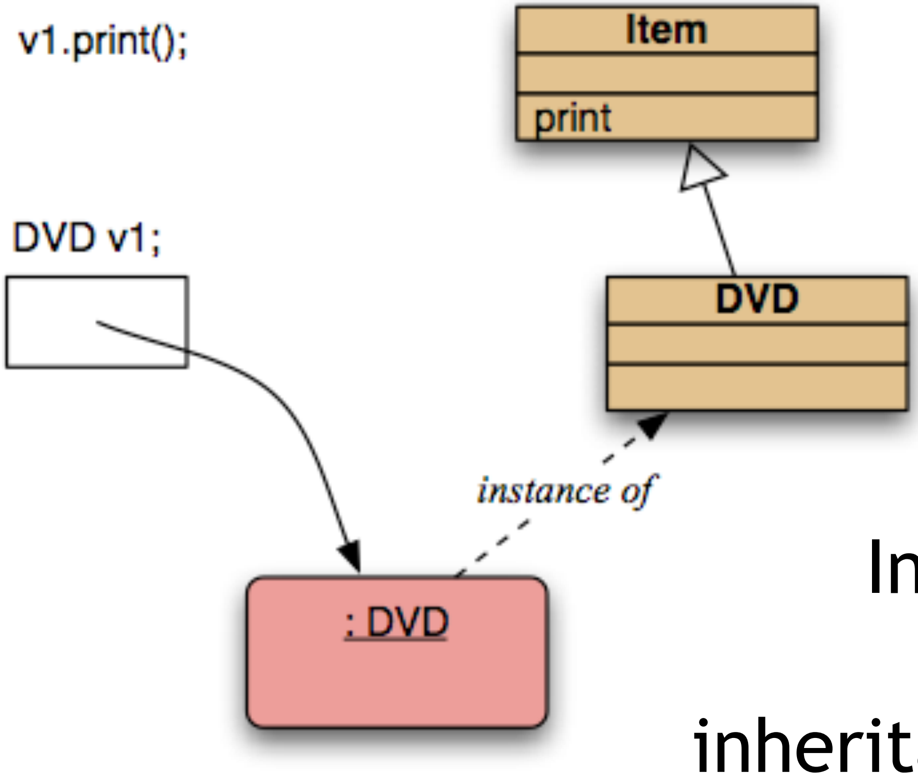
- Superclass and subclass define methods with the same signature.
- Each has access to the fields of its class.
- Superclass satisfies static type check.
- Subclass method is called at runtime – it *overrides* the superclass version.
- What becomes of the superclass version?

Method lookup



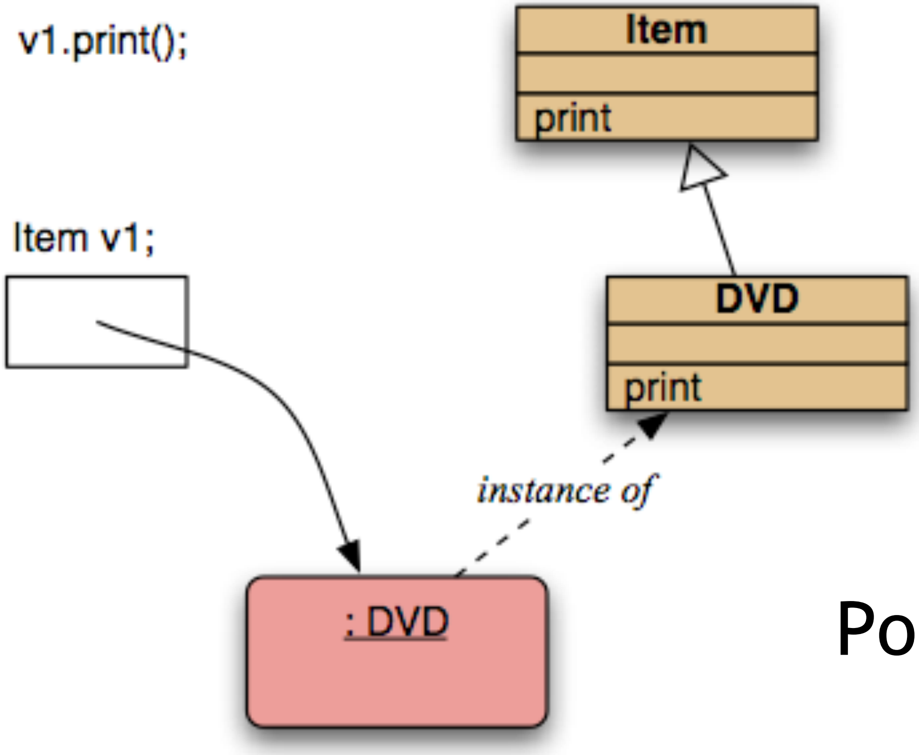
No inheritance or polymorphism.
The obvious method is selected.

Method lookup



Inheritance but no overriding. The inheritance hierarchy is ascended, searching for a match.

Method lookup



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.

Calling an overridden method

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

Calling an overridden method

Contrary to the case of **super** calls in constructors:

- The method name of the superclass method is explicitly stated.
- A **super** call in a method always has the form:

  super.*method-name* (*parameters*) *The parameter list can be empty.*

- The **super** call in methods may occur anywhere within that method. It does not have to be the first statement.
- No automatic **super** call is generated and no **super** call is required; it is entirely optional.

Method polymorphism

- 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.

The Object class's methods

- 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.

Overriding toString

```
public class Item
{
    ...

    public String toString()
    {
        String line1 = title + " (" + playingTime + " mins)");

        if (gotIt)
        {
            return line1 + "*\n" + "    " + comment + "\n");
        }
       else
    {
            return line1 + "\n" + "    " + comment + "\n");
        }
    }
    ...
}
```

Overriding toString

- 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);`

Object equality

- What does it mean for two objects to be ‘the same’?
  - Reference equality.
  - Content equality.
- Compare the use of == with .equals()

Overriding the equals method

```
public boolean equals(Object obj)
{
    if (this == obj)
        return true;

    if (! (obj instanceof ThisType))
        return false;

    ThisType other = (ThisType) obj;

    … compare fields of this and other
}
```

Overriding equals in Student

```
public boolean equals(Object obj)
{
    if (this == obj)
        return true;

    if (! (obj instanceof Student))
        return false;

    Student other = (Student) obj;

    return name.equals(other.name) &&
           id.equals(other.id) &&
           credits == other.credits;
}
```

Overriding hashCode in Student

two objects that are the same as determined by a call to **equals** must return identical values from **hashCode**.

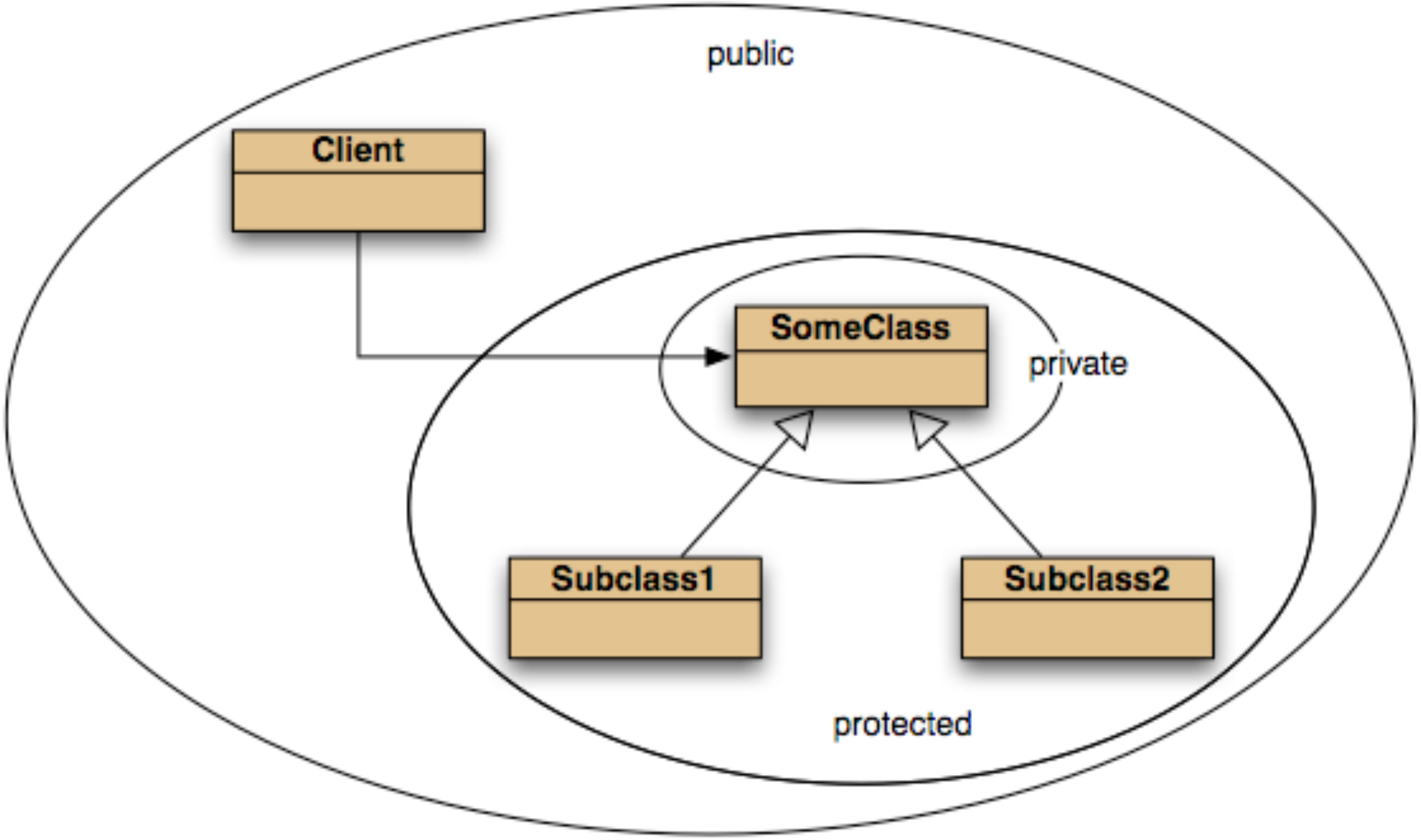
```
/**
 * Hashcode technique taken from
 * Effective Java by Joshua Bloch.
 */
public int hashCode()
{
    int result = 17;
    result = 37 * result + name.hashCode();
    result = 37 * result + id.hashCode();
    result = 37 * result + credits;
    return result;
}
```

This is beyond the scope of this subject!

Protected access

- 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.

Access levels



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.