

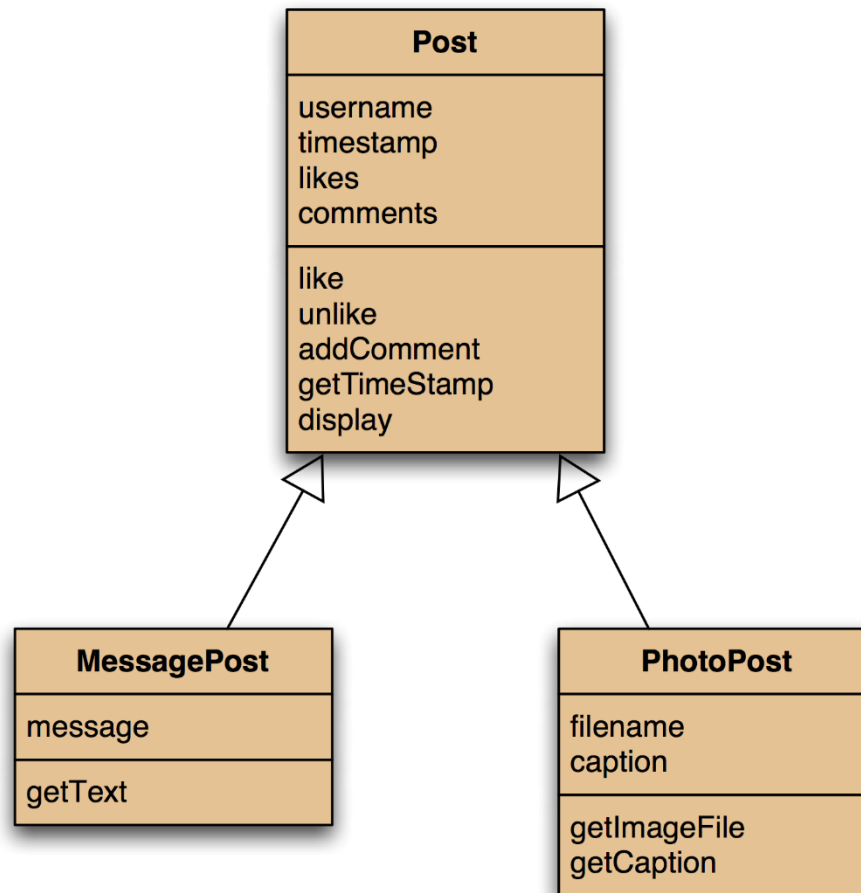


# More about inheritance

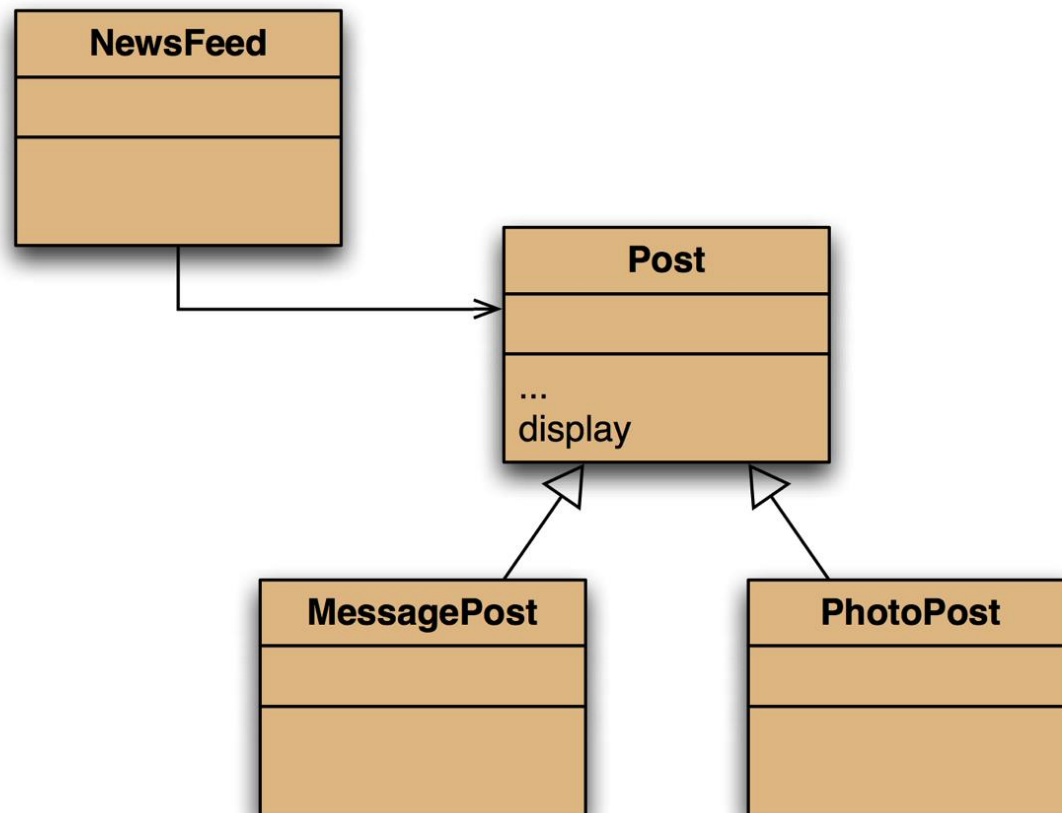
## Exploring polymorphism

Edited by Ehsan Edalat

# The Class Diagram



# The inheritance hierarchy



# Conflicting output

Leonardo da Vinci

Had a great idea this morning.

But now I forgot what it was. Something to do with flying ...

40 seconds ago - 2 people like this.

No comments.

Alexander Graham Bell

[experiment.jpg]

I think I might call this thing 'telephone'.

12 minutes ago - 4 people like this.

No comments.

What we want

Leonardo da Vinci

40 seconds ago - 2 people like this.

No comments.

Alexander Graham Bell

12 minutes ago - 4 people like this.

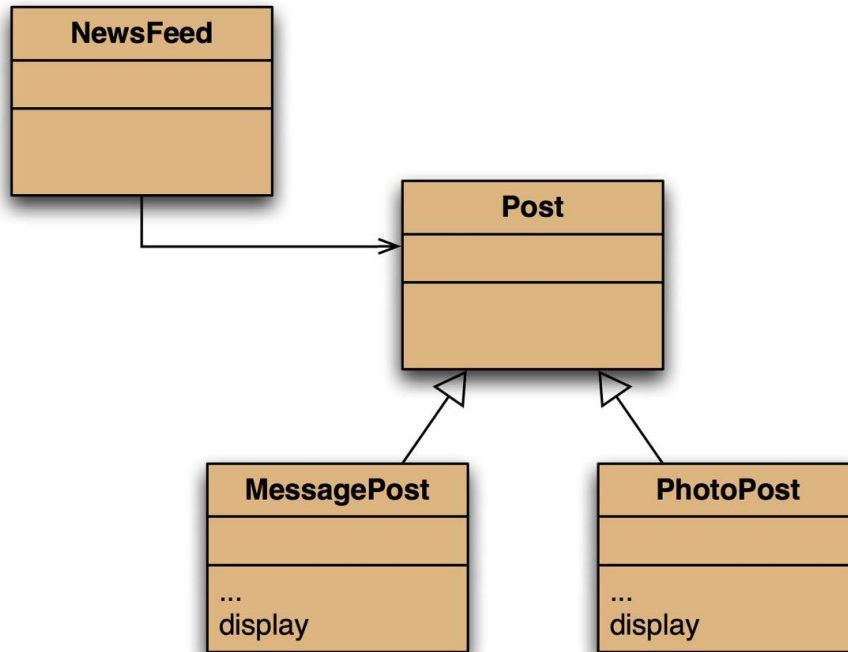
No comments.

What we have

# The problem

- The `display` method in `Post` 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 **display** where it has access to the information it needs.

Each subclass has its own version.

But **Post**'s fields are **private**.

**NewsFeed** cannot **find** a **display** method in **Post**.



# 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?

```
Vehicle v1 = new 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 (Post post : posts) {  
    post.display();    // Compile-time error.  
}
```

# Solution: using instanceof

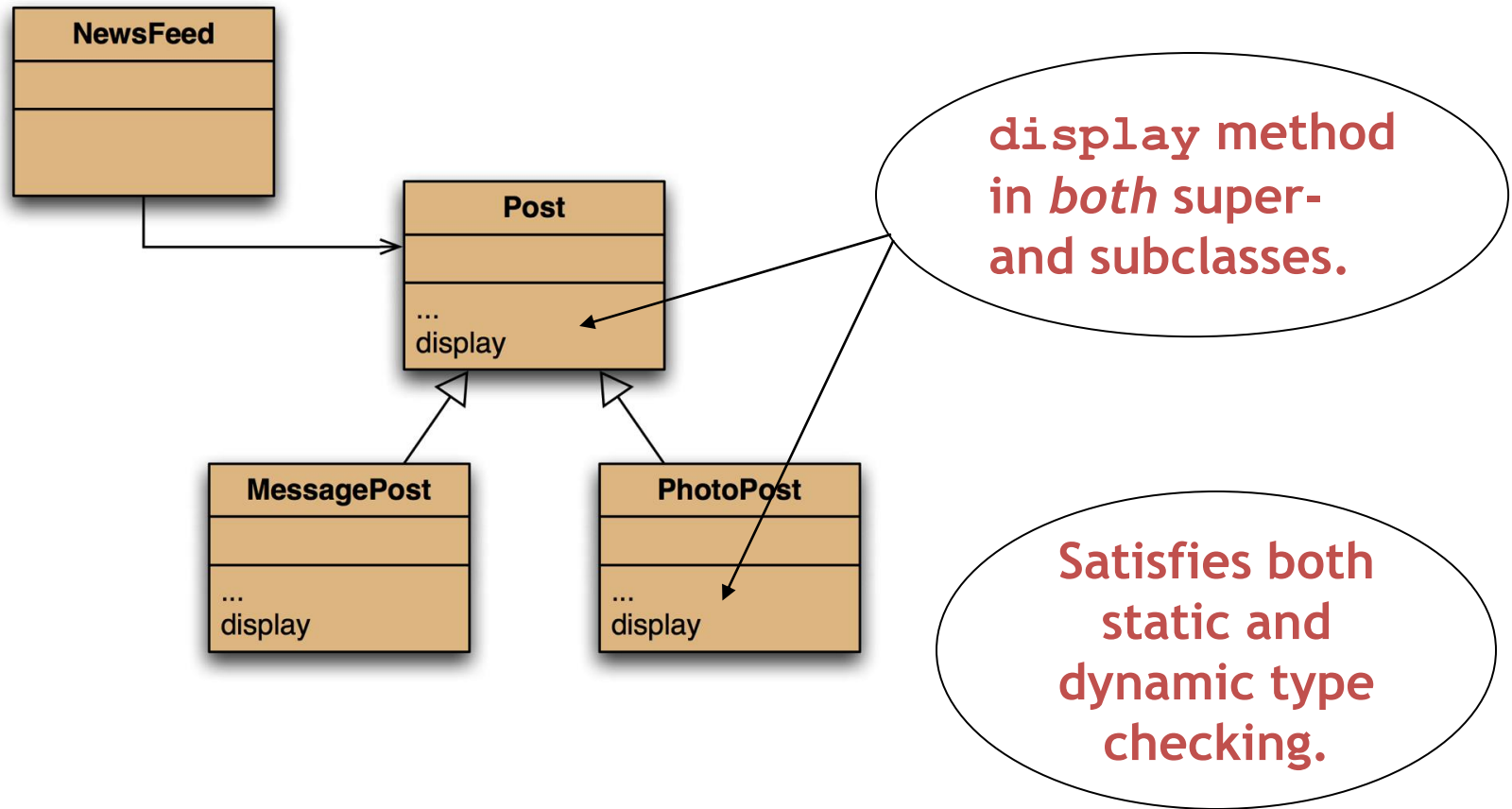
```
public class NewsFeed {  
    ...  
    public void show() {  
        for (Post post : posts) {  
            if (post instanceof MessagePost) {  
                MessagePost msg = (MessagePost) post;  
                msg.display();  
            } else if (post instanceof PhotoPost) {  
                PhotoPost photo = (PhotoPost) post;  
                photo.display();  
            }  
            System.out.println();  
        }  
    }  
    ...  
}
```

# The `instanceof` operator

- Used to determine the dynamic type.
- Recovers ‘lost’ type information.
- Usually precedes assignment with a cast to the dynamic type:

```
if (post instanceof MessagePost) {  
    MessagePost msg = (MessagePost) post;  
    ... access MessagePost methods via msg ...  
}
```

# Overriding: the better solution



# 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?

# Distinct static and dynamic types

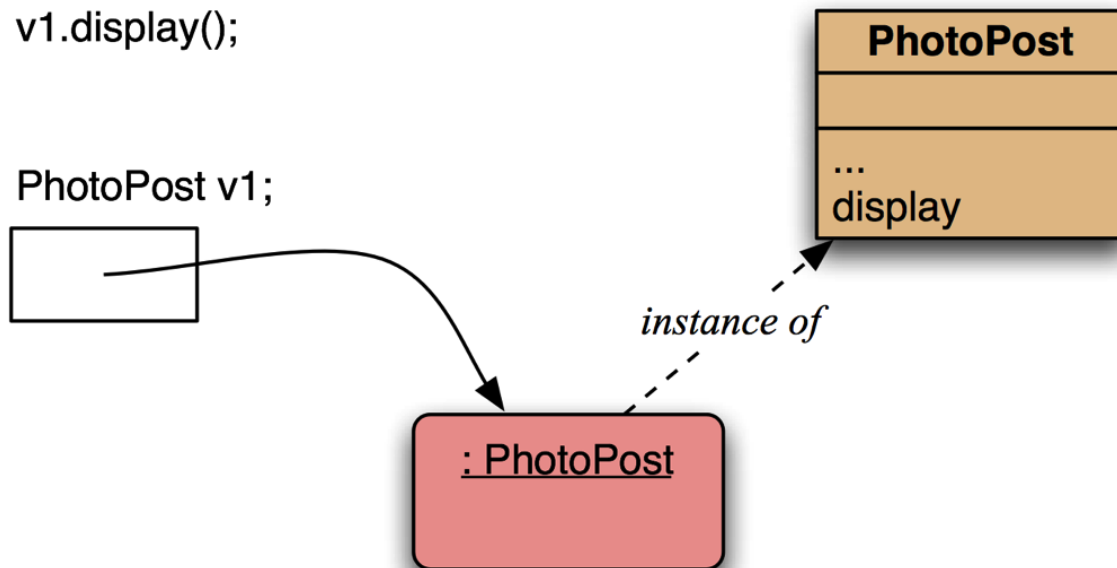




# Method lookup

v1.display();

PhotoPost v1;

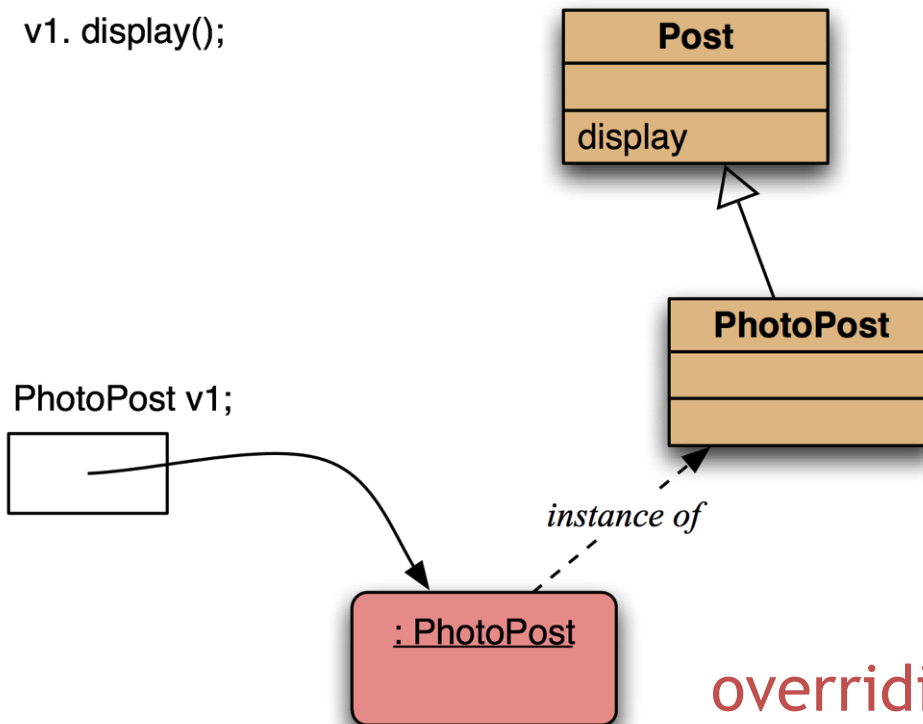


No inheritance or polymorphism.  
The obvious method is selected.



# Method lookup

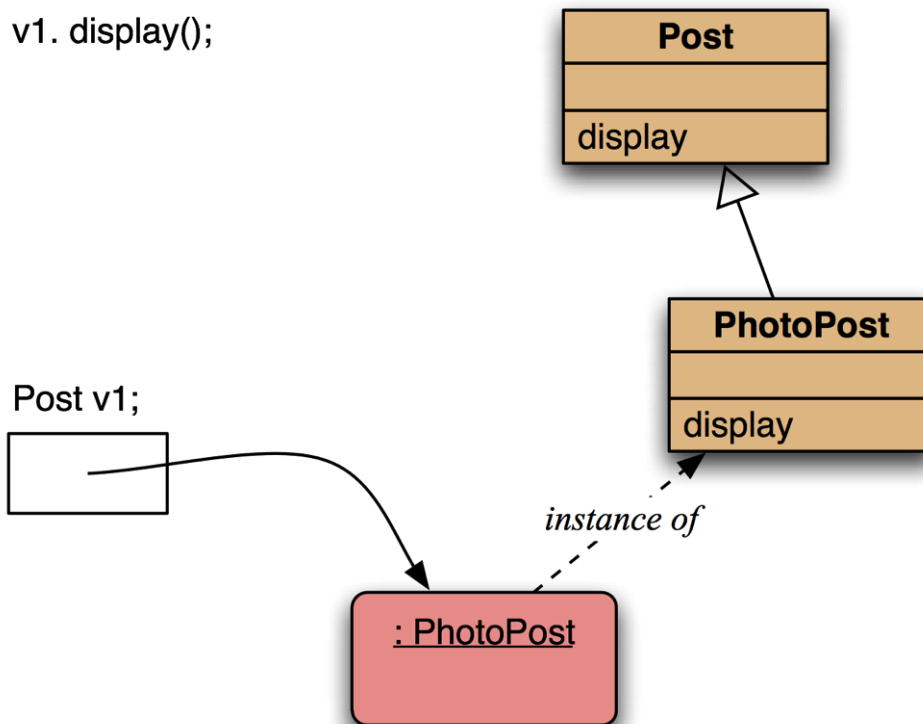
v1. display();



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

# Method lookup

v1. display();



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, the superclass is searched.
- This is repeated until a match is found, or the class hierarchy is exhausted.
- Overriding methods take precedence - they override inherited copies.

# 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 void display()  
{  
    super.display();  
    System.out.println(" [" +  
                        filename +  
                        "]" );  
    System.out.println(" " + caption);  
}
```



# Method polymorphism

- A **polymorphic variable** can store objects of varying types.
- We have been discussing *polymorphic method dispatch*.
- 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 in Post

```
public String toString()
{
    String text = username + "\n" +
                    timeStamp();
    if(likes > 0) {
        text += " - " + likes + " people like this.\n";
    }
    else {
        text += "\n";
    }
    if(comments.isEmpty()) {
        return text + " No comments.\n";
    }
    else {
        return text + " " + comments.size() +
                    " comment(s). Click here to view.\n";
    }
}
```

# Overriding toString

- Explicit print methods can often be omitted from a class:  
`System.out.println(post.toString()) ;`
- Calls to `println` with just an object automatically result in `toString` being called:

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

# StringBuilder

- Consider using **StringBuilder** as an alternative to concatenation:

```
StringBuilder builder = new  
StringBuilder();  
builder.append(username);  
builder.append('\n');  
builder.append(timestamp);  
...  
return builder.toString();
```



# Object equality

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

# Overriding equals

```
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

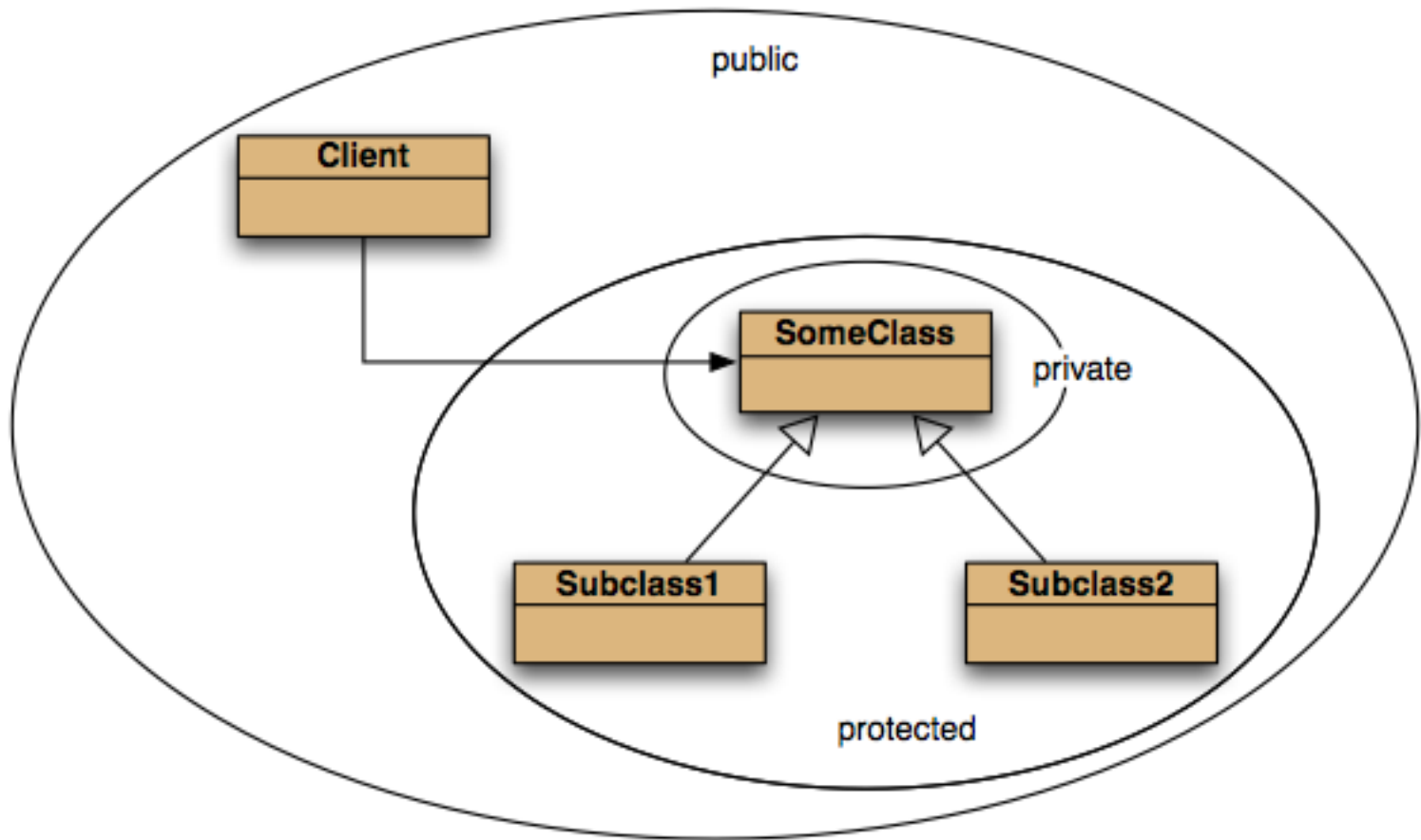
```
/**
 * Hashcode technique taken from
 * Effective Java by Joshua Bloch.
 */
public int hashCode()
{
    int result = 17;
    result = 37 * result + name.hashCode();
    result = 37 * result + id.hashCode();
    result = 37 * result + credits;
    return result;
}
```



# 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



# Let's Do Some Exercises!

- Read another example: Section 11.11
- Work with the dome project and complete it with the different implementation of print method. (instanceof, overriding, and protected members)

**Exercise 11.11** Assume that you see the following lines of code:

```
Device dev = new Printer();  
dev.getName();
```

**Printer** is a subclass of **Device**. Which of these classes must have a definition of method **getName** for this code to compile?

**Exercise 11.12** In the same situation as in the previous exercise, if both classes have an implementation of **getName**, which one will be executed?



# 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.