Inheritance

Exploring Polymorphism

Produced by: Dr. Siobhán Drohan
Mr. Colm Dunphy
Mr. Diarmuid O'Connor
Dr. Frank Walsh

Waterford Institute of Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

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

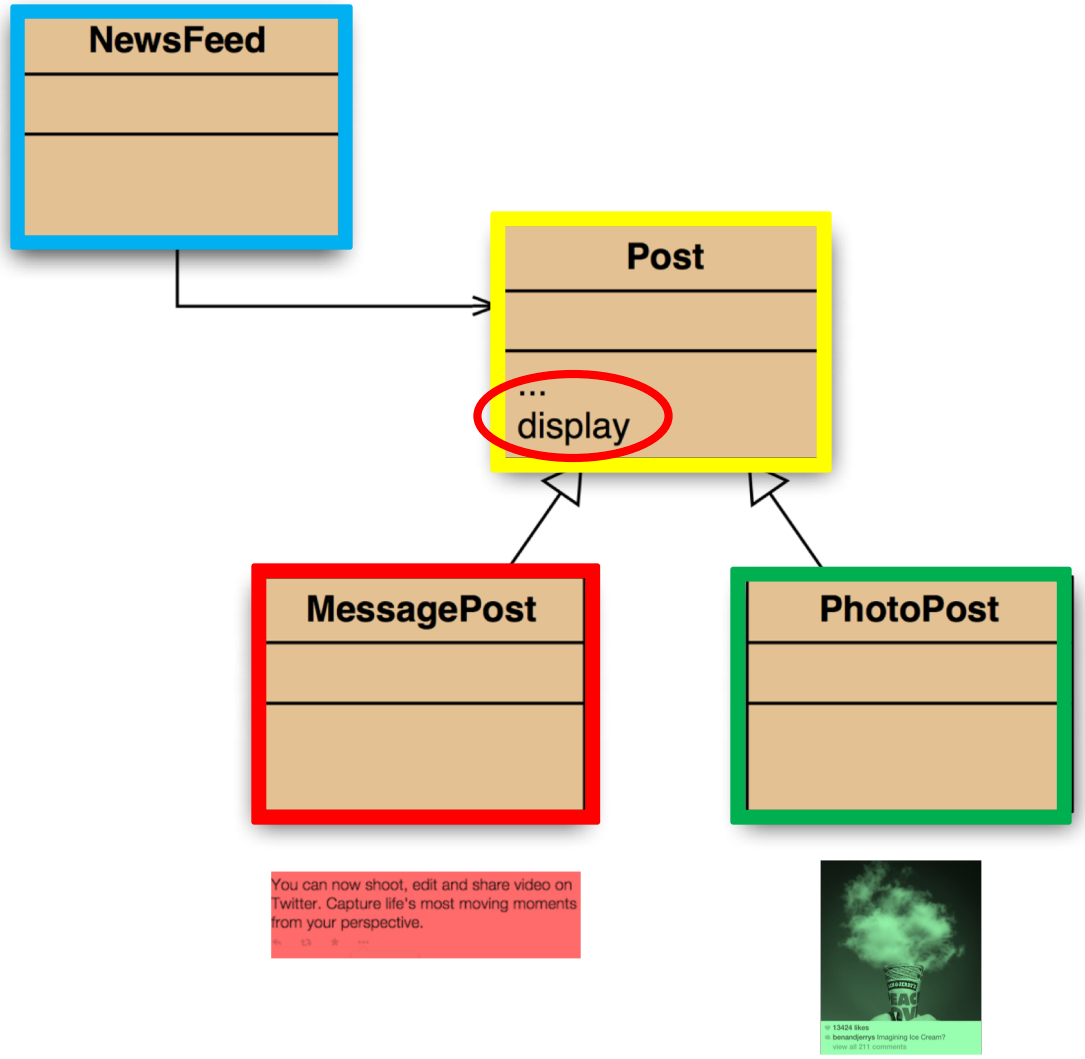
Lectures and Labs

- This weeks lectures and labs are based on examples in:
  - **Objects First with Java** - A Practical Introduction using BlueJ, © David J. Barnes, Michael Kölling (https://www.bluej.org/objects-first/)

Topic List

1. Method polymorphism
   - e.g. display()
2. Static and dynamic type
3. Overriding
4. Dynamic method lookup
5. Protected access

Social NetworkV2 – Inheritance Hierarchy



Testing the **display()** method…

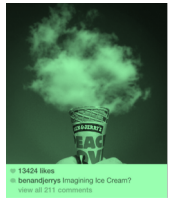
Create this **MessagePost**

You can now shoot, edit and share video on Twitter. Capture life's most moving moments from your perspective.

username	Leonardo da Vinci
message	Had a great idea this morning. But now I forgot what it was. Something to do with flying ...
likes	40 seconds ago - 2 people **like** this.
comments	**No comments.**

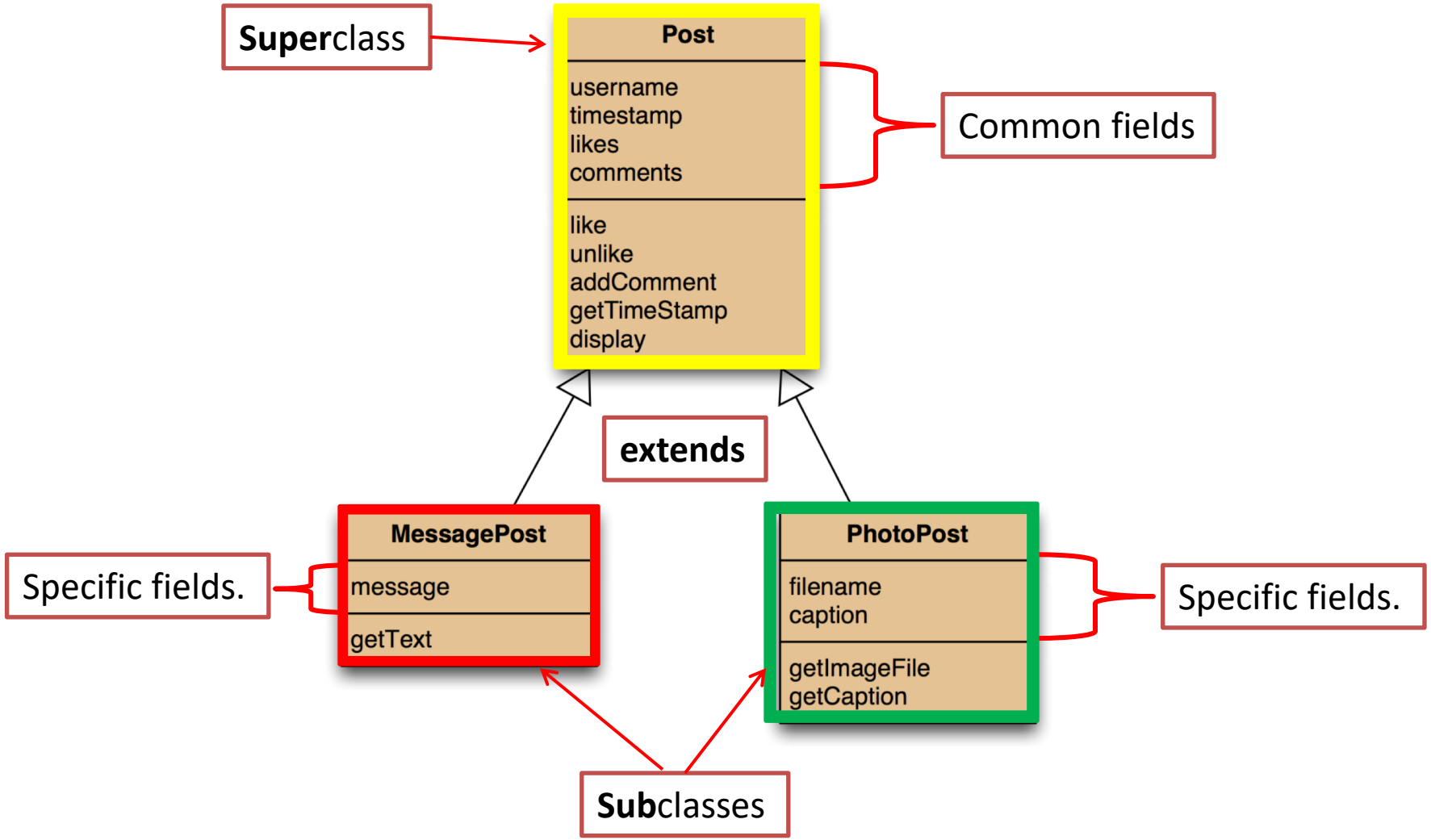
Testing the **display()** method…

Create this **PhotoPost**



username	Alexander Graham Bell
filename	[experiment.jpg]
caption	I think I might call this thing 'telephone'.
likes	12 minutes ago - 4 people **like** this.
comments	**No comments.**

RECAP:
Social Network V2 - Using inheritance



Testing the **display()** method…

```
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**

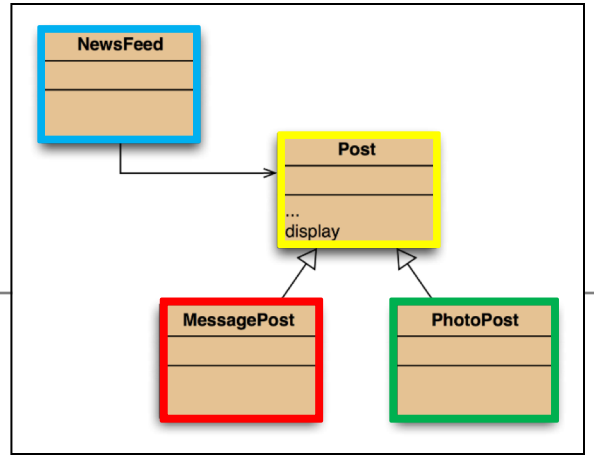
```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

What we **have**

message **filename** **caption** are missing from what we have. **i.e. the subclass specific fields**

The problem



- The **`display()`** method in **`Post`** only prints the <u>common fields</u>.

- 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**?

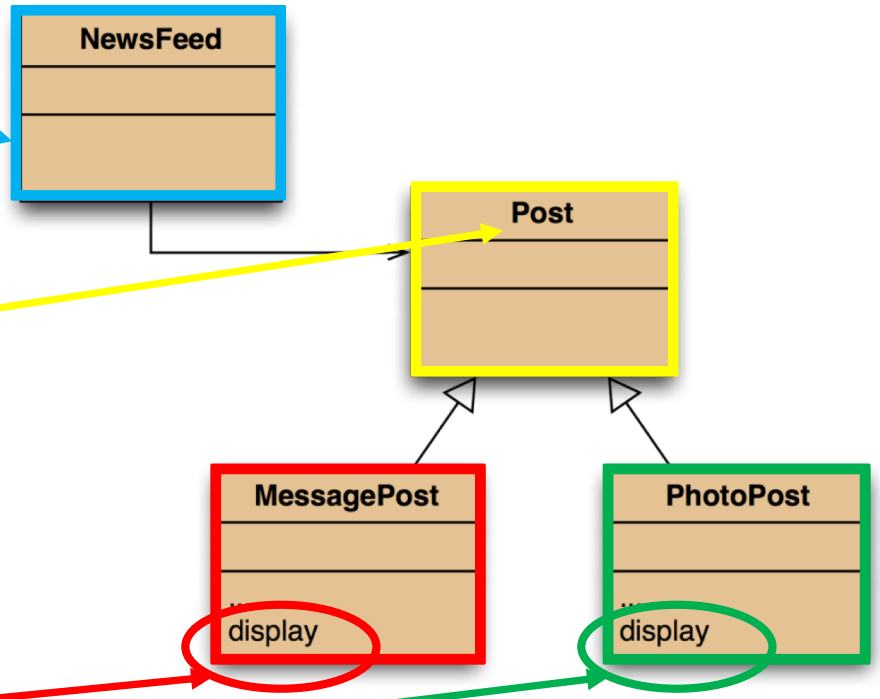
3) `NewsFeed`

cannot find a **`display()`** method in `Post`.

2) But `Post`'s fields are private.

1) Place a **`display()`**
where it has access to the information it needs.

- i.e. in each subclass
  - One version for **MessagePost**
  - One version for **PhotoPost**



Topic List

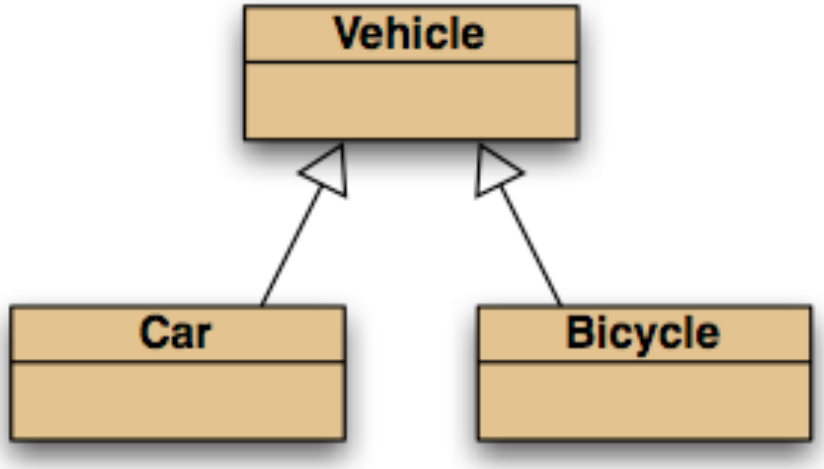
1. Method polymorphism
   - E.g. display()
2. Static and dynamic type
3. Overriding
4. Dynamic method lookup
5. Protected access

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**

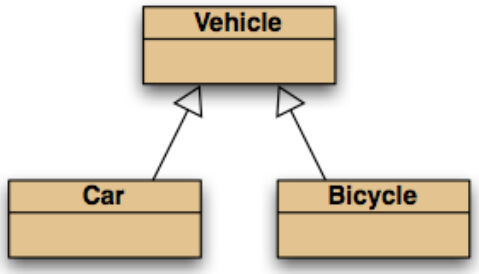
Lets revisit our vehicle example…



***sub**class objects*
may be assigned to
***super**class variables*

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

Static and dynamic **type**



What is the type of c1?

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

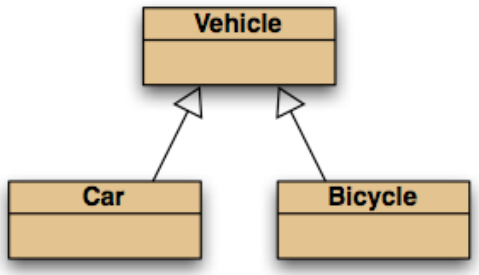
The declared type of a variable is its ***static*** *type*.

What is the type of v1?

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

The type of the object a variable refers to is its ***dynamic*** *type*.

Static and dynamic **type**



The compiler's job is to check for static-type violations.

What is the type of **v1**?

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

The declared type of a variable is its *static type*.

The type of the object a variable refers to is its *dynamic type*.

Recall our attempt to solve this problem…

```
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**

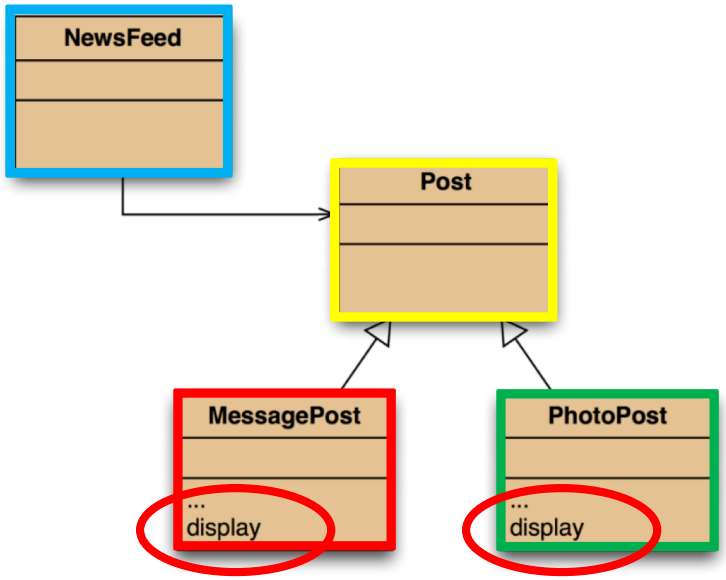
```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

Alexander Graham Bell
12 minutes ago  -  4 people like this.
  No comments.
```

What we **have**

message **filename** **caption** are missing from what we have. **i.e. the subclass specific fields**

Recall our attempt to solve this problem…



We placed **`display()`** in each subclass where it has access to the information it needs.

But:

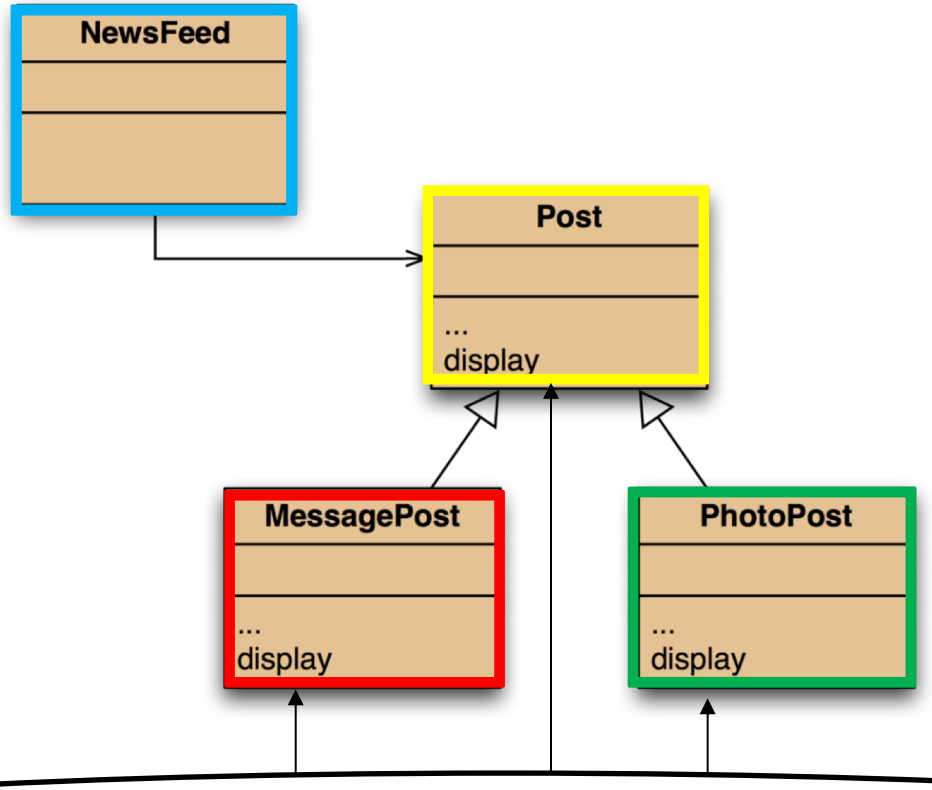
- `Post`'s fields are private and `NewsFeed` cannot find a **`display()`** method in `Post`.

```
for(Post post : posts) {
  post.display();
                  // But there is no display() method in Post
                  //
                  // Compile-time error (static-type violation)
                  //
                  // because method display() is not found
                  // in the Post class
  }
```

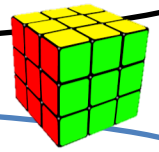
Topic List

1. Method polymorphism
   - display()
2. Static and dynamic type
3. **Overriding**
4. Dynamic method lookup
5. Protected access

Overriding - the solution to our problem



`display()` method in ***both*** <u>super</u> and <u>sub</u> classes.



Satisfies both static and dynamic **type** checking.

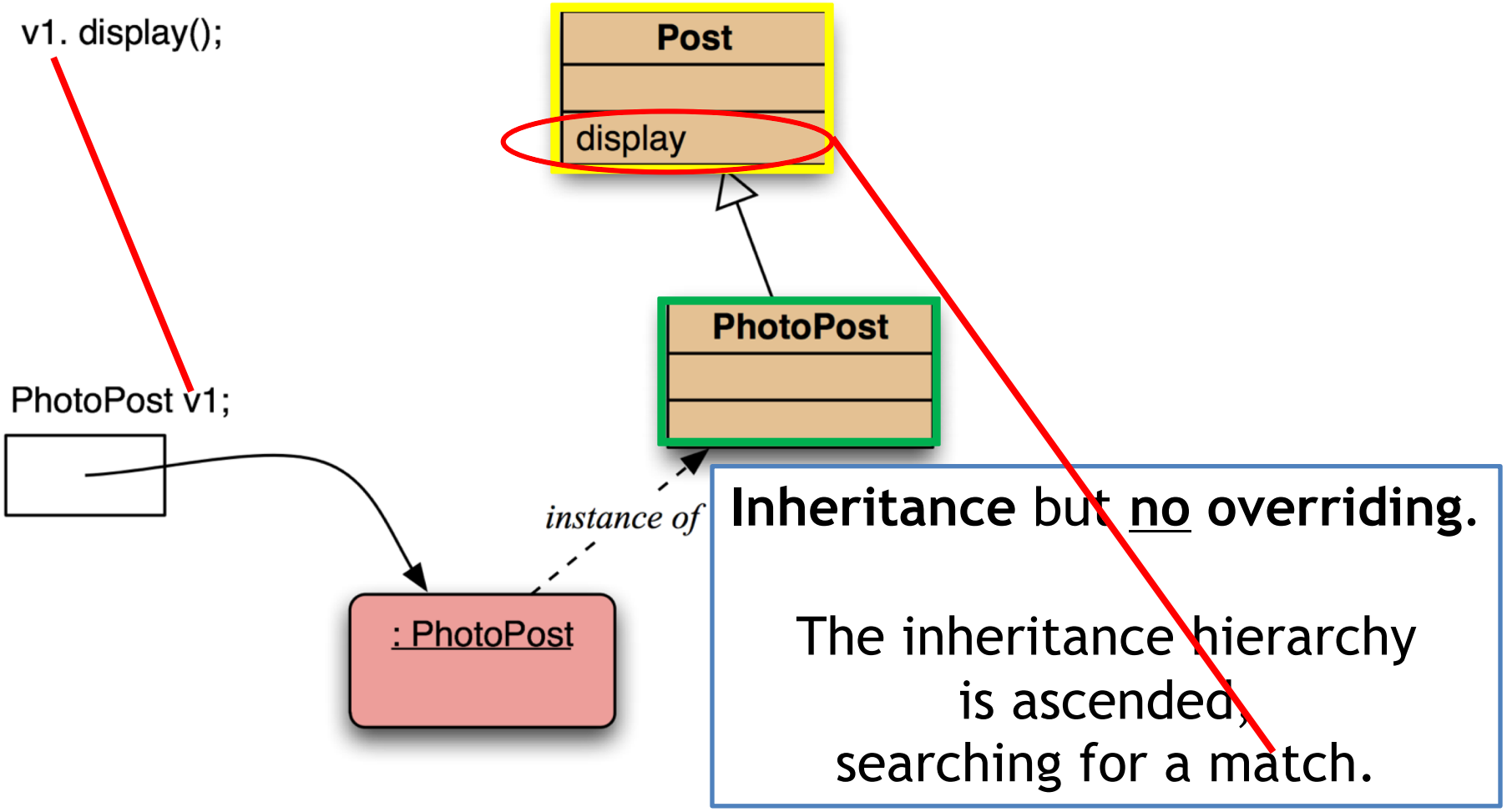
Overriding

- **Super**class and **sub**class define methods
  - with the same signature.
- Each has
  - access to the fields of its class.
- **Super**class satisfies **static type check**.
- **Sub**class method is called at runtime
  - it ***overrides*** the superclass version.
- What becomes of the superclass version?
  - Lets see…

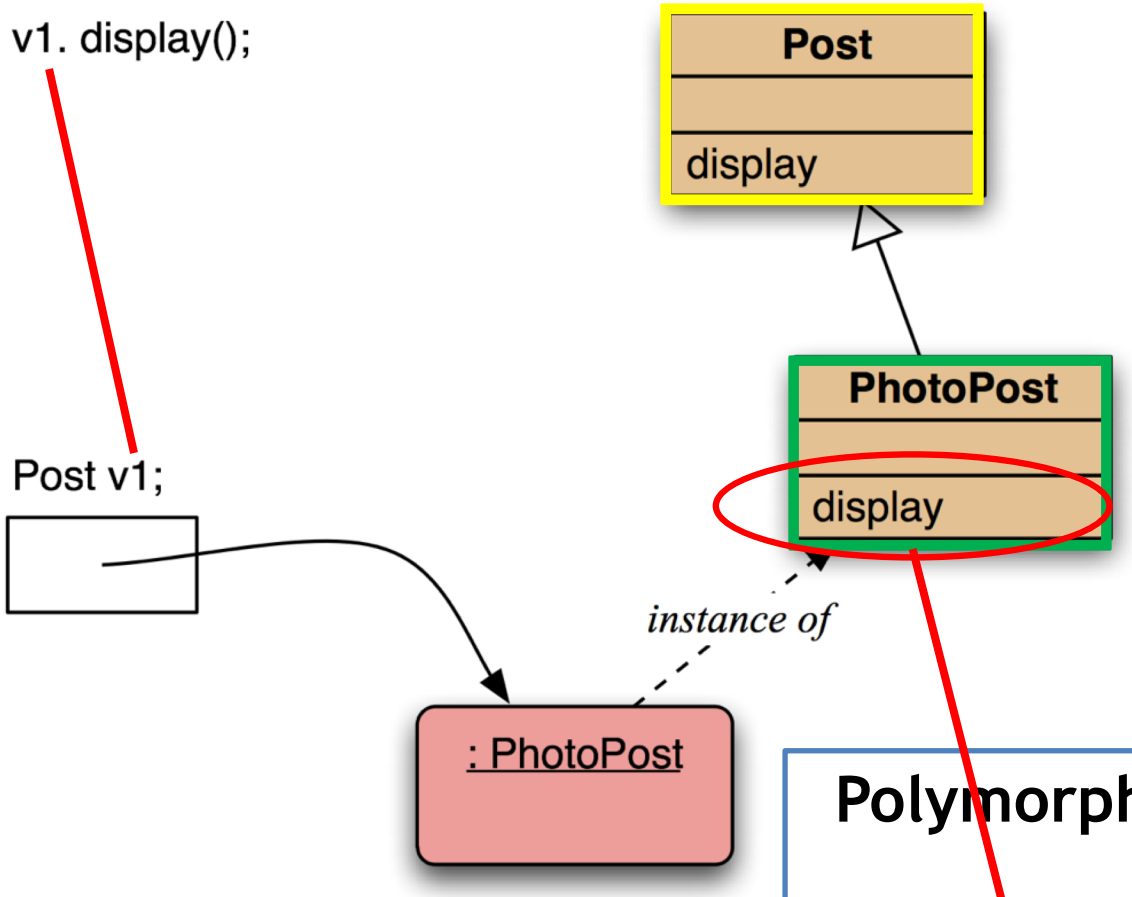
Topic List

1. Method polymorphism
   - display()
2. Static and dynamic type
3. Overriding
4. Dynamic method lookup
5. Protected access

Dynamic method lookup



Dynamic method lookup



Polymorphism and overriding.

The ‘first’ version found is used.

Dynamic method lookup summary

1. The variable is accessed.
2. The object stored in the variable is found.
3. The class of the object is found.
4. The class is searched for a method match.
5. If no match is found, the superclass is searched.
6. This is repeated until a match is found, or the class hierarchy is exhausted.
7. Overriding methods take precedence
   - i.e. **stop when you find a match**.

Super call in methods

- Overridden methods are hidden
  - but we often still want to be able to call them explicitly.

- An overridden method
  can be called from the method that overrides it.
  - `super.method(...)`
  - Recall we used **`super`** in our constructors.

e.g. calling an overridden method

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

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 `instanceof` operator

`instanceof` is used to determine the **dynamic type**.

- It can recover 'lost' type information.
- It usually precedes assignment with a **cast** to the **dynamic type**:

```
if (post instanceof MessagePost) {

    MessagePost msg = (MessagePost) post;

    … e.g. then access MessagePost methods via msg …
}
```

Recall the Object class…

java.lang

Class Object

java.lang.Object

```
public class Object
```

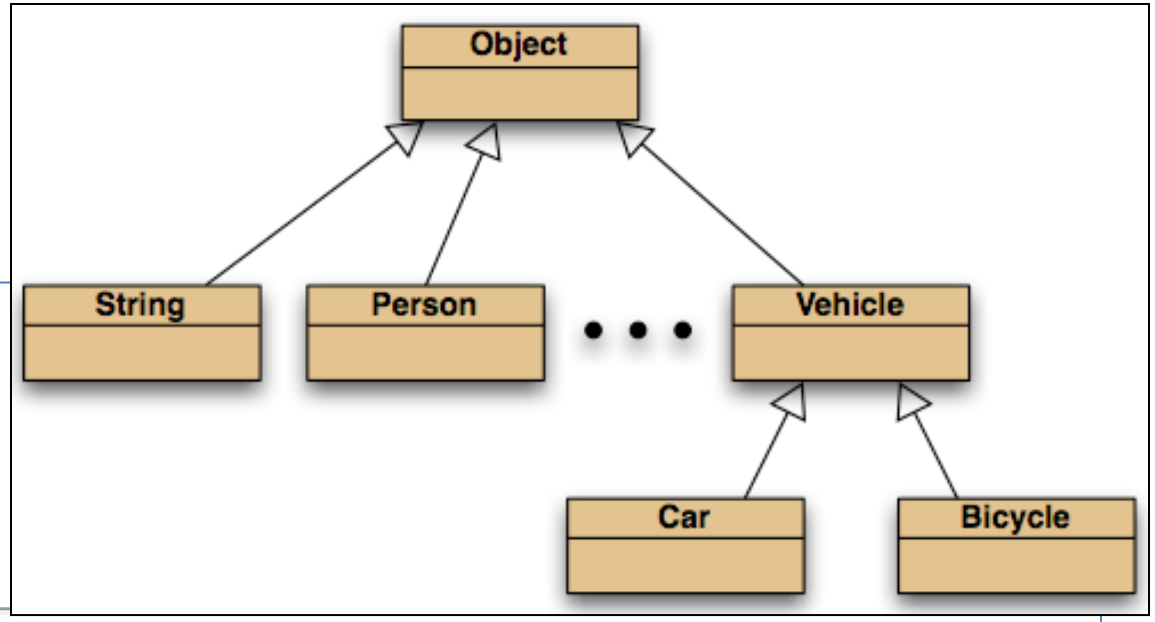
Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

Recall the Object class…

***All** classes inherit from* **Object**.



java.lang

Class Object

java.lang.Object

```
public class Object
```

Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

Methods in **`Object`** are inherited by all classes.

Any of these may be overridden.

Methods

Modifier and Type	Method and Description
`protected Object`	**`clone()`** Creates and returns a copy of this object.
`boolean`	**`equals(Object obj)`** Indicates whether some other object is "equal to" this one.
`protected void`	**`finalize()`** Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
`Class<?>`	**`getClass()`** Returns the runtime class of this `Object`.
`int`	**`hashCode()`** Returns a hash code value for the object.
`void`	**`notify()`** Wakes up a single thread that is waiting on this object's monitor.
`void`	**`notifyAll()`** Wakes up all threads that are waiting on this object's monitor.
`String`	**`toString()`** Returns a string representation of the object.
`void`	**`wait()`** Causes the current thread to wait until another thread invokes the **`notify()`** method or the **`notifyAll()`** method for this object.
`void`	**`wait(long timeout)`** Causes the current thread to wait until either another thread invokes the **`notify()`** method or the **`notifyAll()`** method for this object, or a specified amount of time has elapsed.
`void`	**`wait(long timeout, int nanos)`** Causes the current thread to wait until another thread invokes the **`notify()`** method or the **`notifyAll()`** method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
	her thread invokes the this object, or a
	ead invokes the this object, or some tain amount of real time has elapsed.

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" + timeString(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);
```

- We've seen how we can override how the object is printed by creating a **`toString()`** method

Topic List

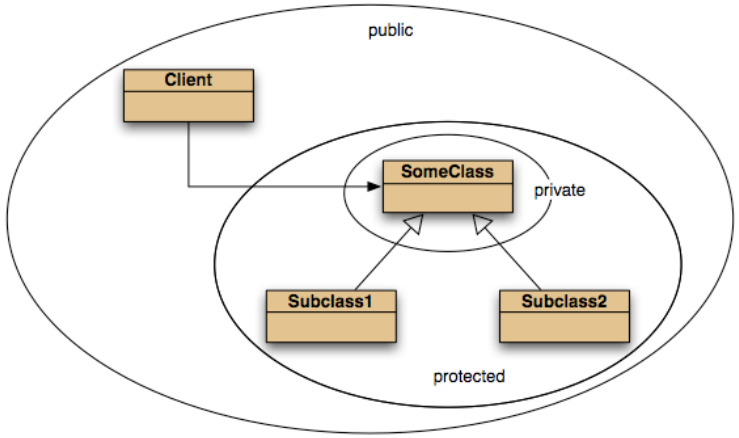
1. Method polymorphism
   - display()
2. Static and dynamic type
3. Overriding
4. Dynamic method lookup
5. Protected access

Protected access

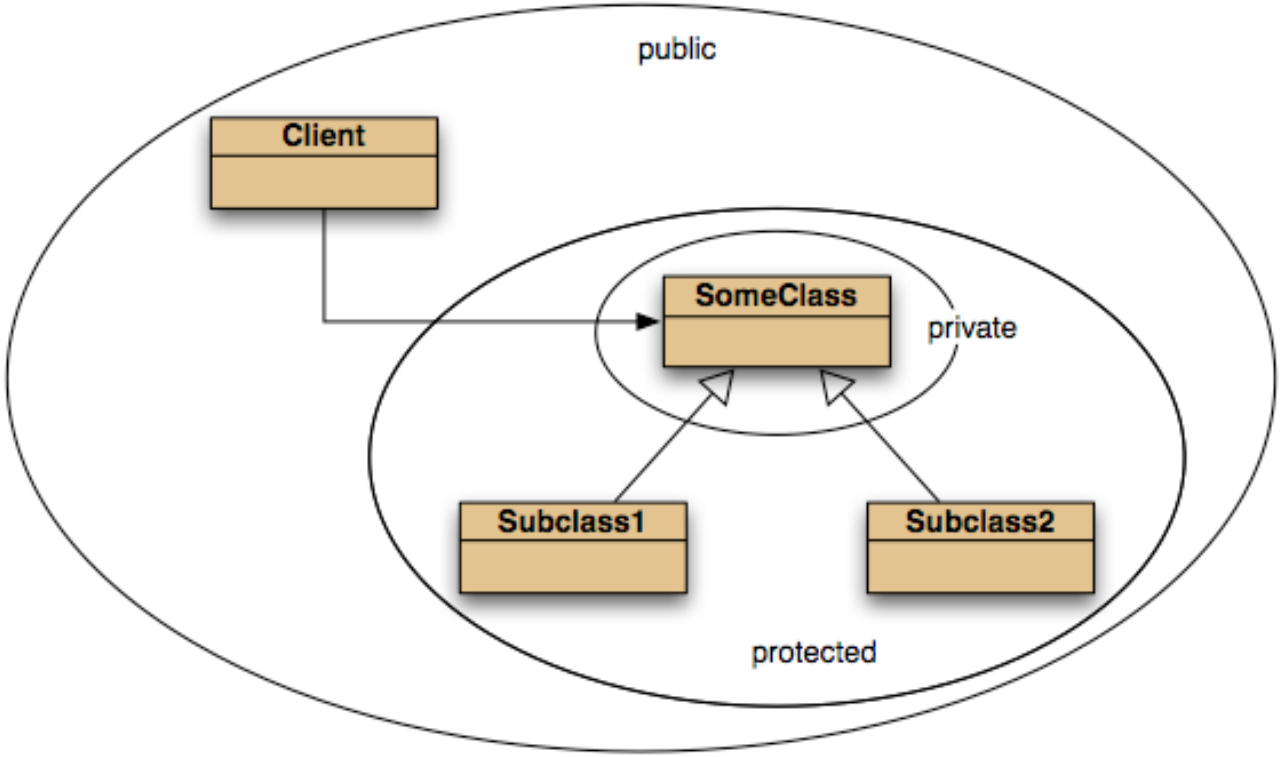
- *Private* access in the superclass
  - may be too restrictive for a subclass
    - Only methods of the class can access the fields.
    - Subclass methods can't

- **Inheritance** is supported by ***protected*** access.
  - Subclass methods can access the fields of the class they inherit from

- ***Protected*** access is
  - more restricted than *public* access.



Access levels



public – all methods in all classes have access
private – only methods in that class have access
protected – only methods in that class, and subclasses have access

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.

Any Questions?