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/

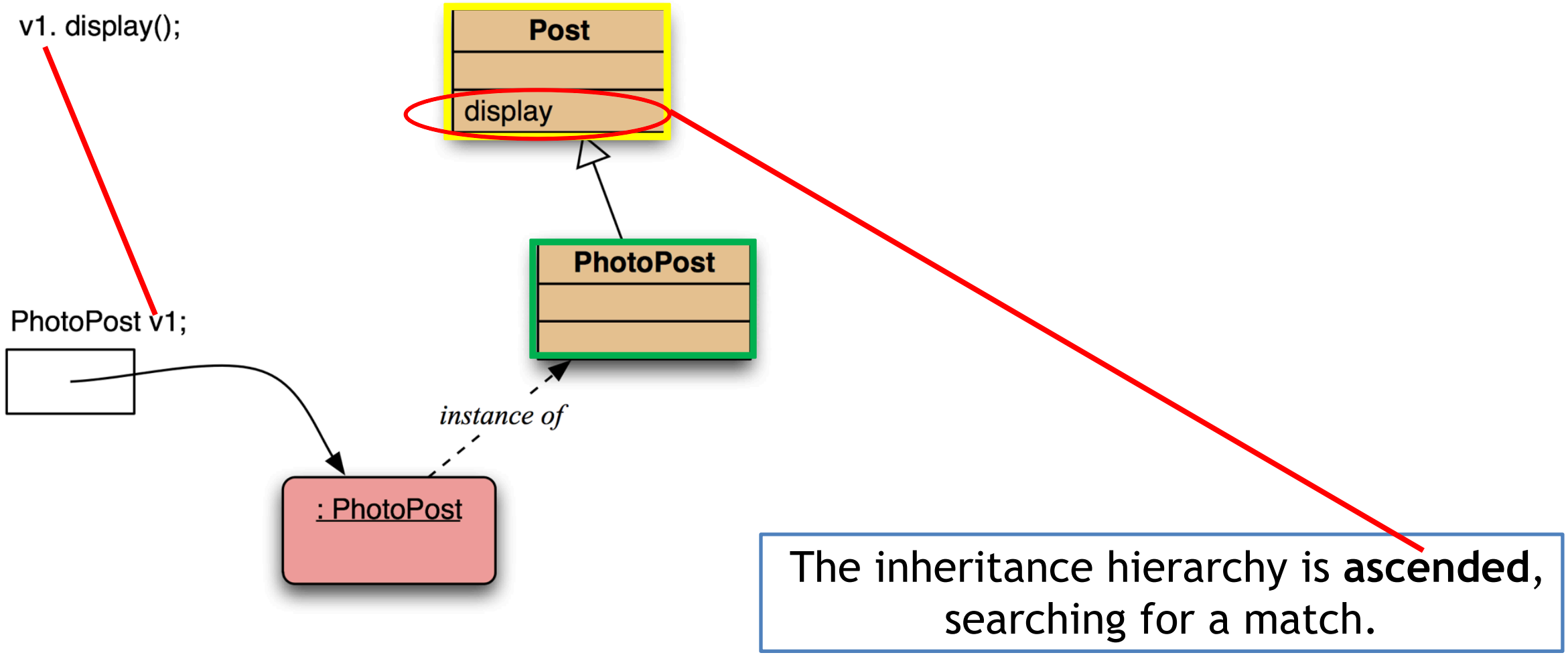
Lectures and Labs

- 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/)

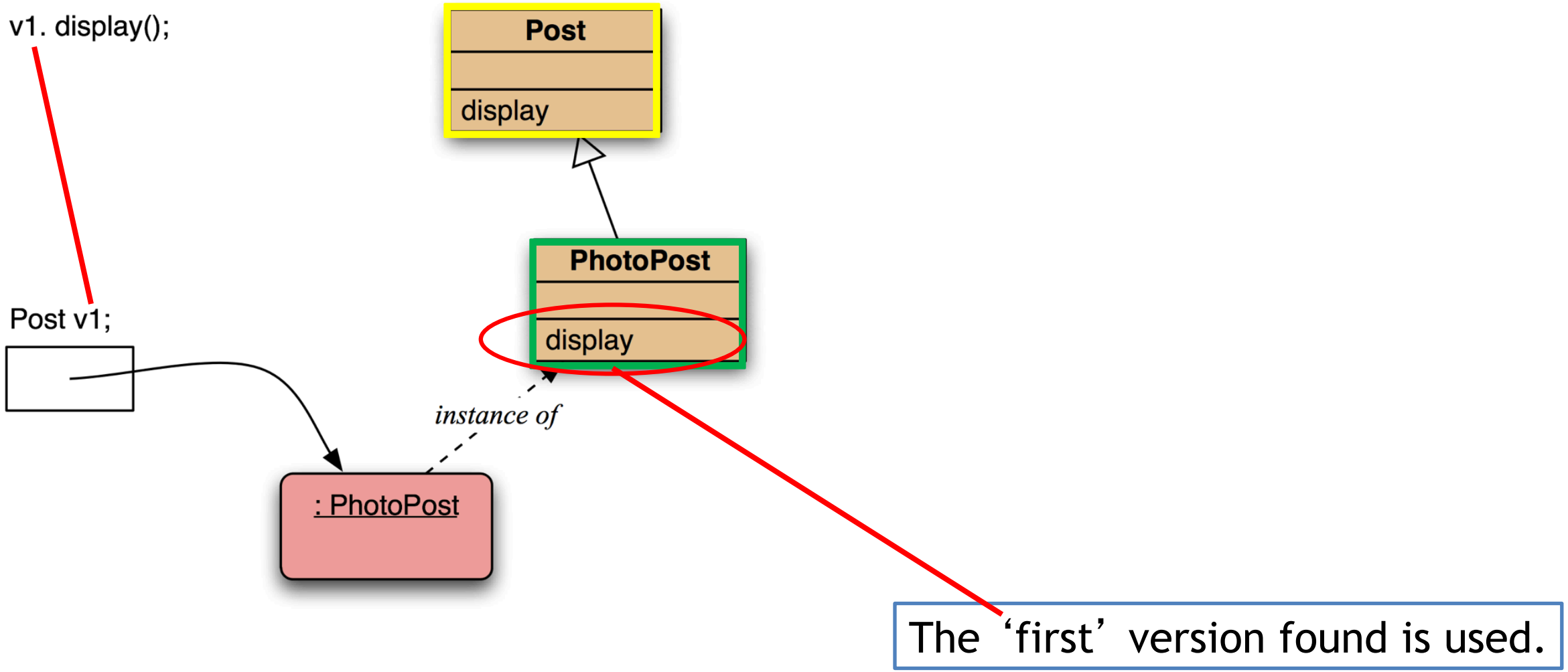
Topic List

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

Dynamic method lookup
1) **Inheritance but no overriding**



Dynamic method lookup
2) **Polymorphism and overriding.**



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`**

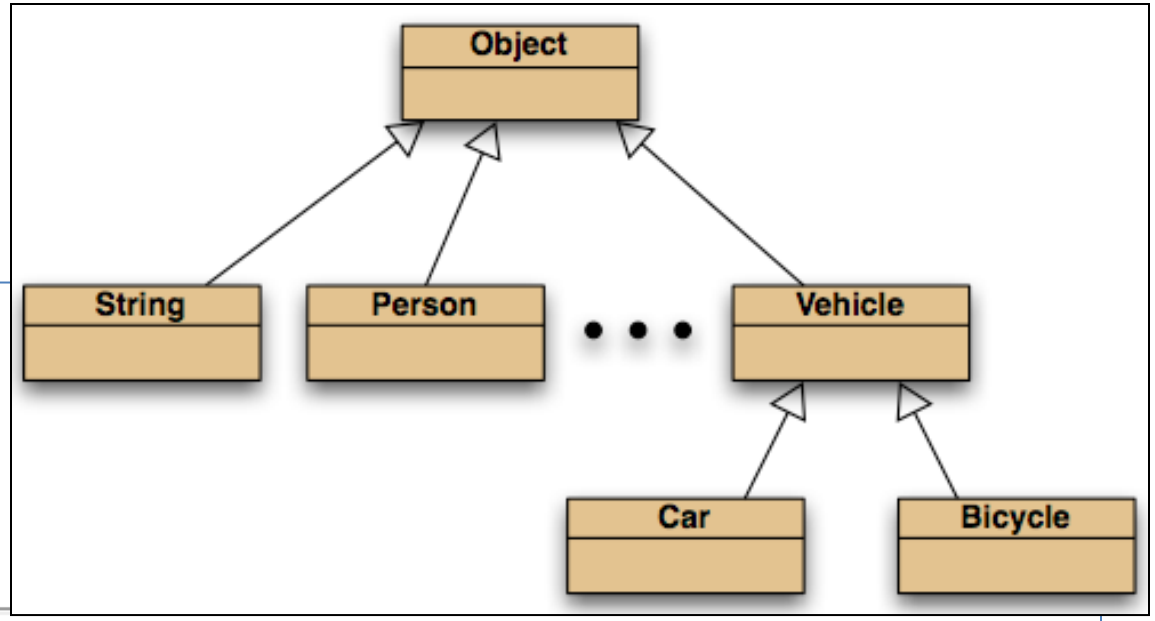
Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

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 this object.

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

Any Questions?