Inheritance

Exploring Polymorphism

Produced by: Ms. Mairéad Meagher
Dr. Siobhán Drohan
Ms Siobhan Roche

Ollscoil Teicneolaíochta an Oirdheiscirt
South East Technological University

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

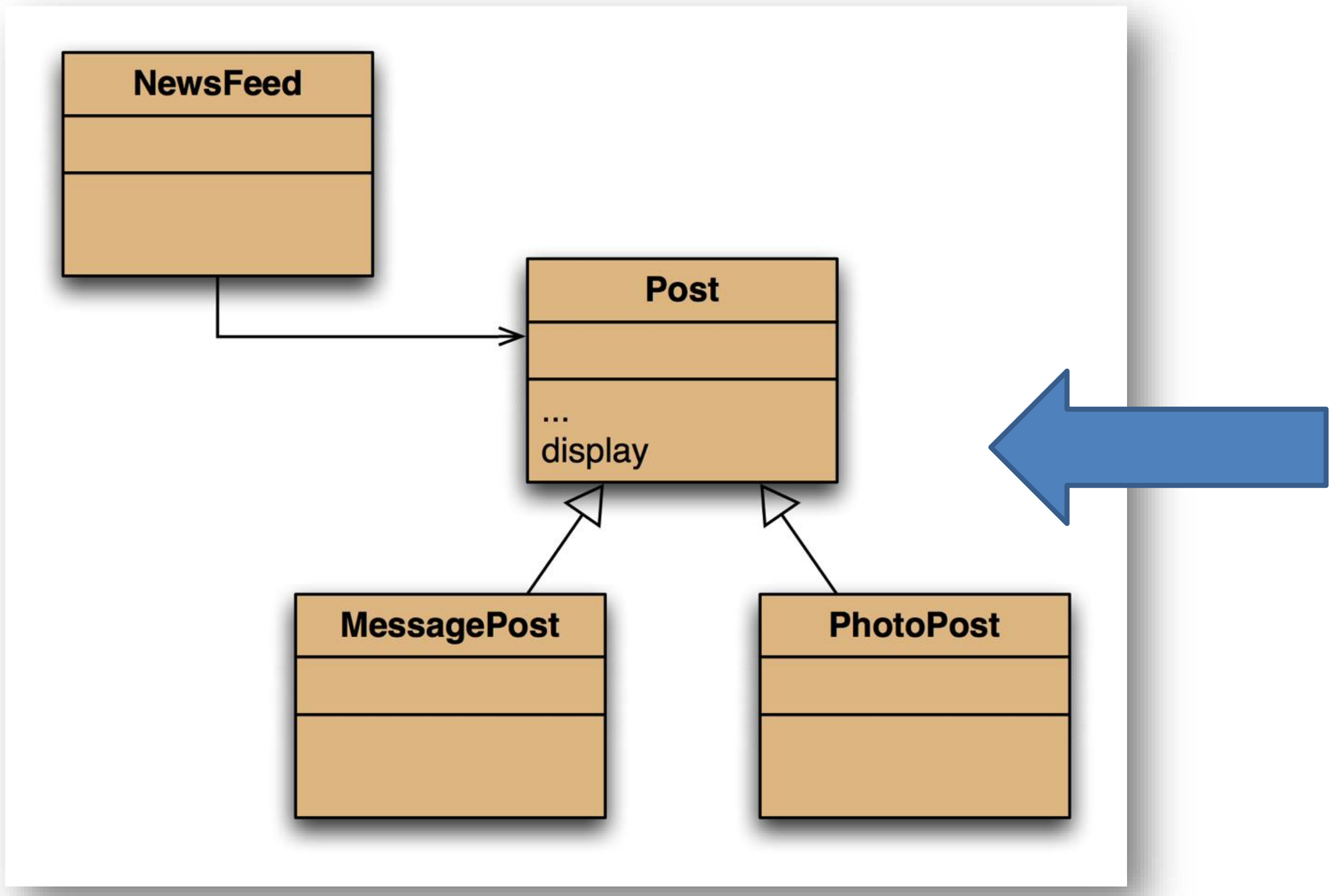
Lectures and Labs

- 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/)

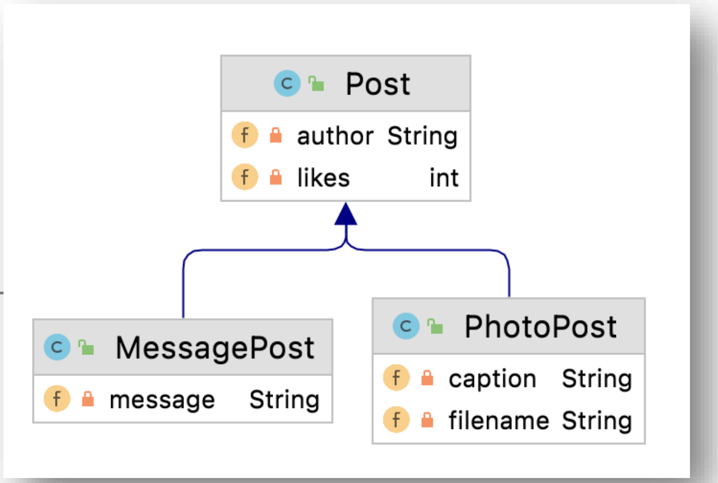
Topic List

- Method polymorphism
- Static and dynamic type
- Overriding
- Dynamic method lookup
- Protected access

Social Network V5.0 – Inheritance Hierarchy



Testing the display method...



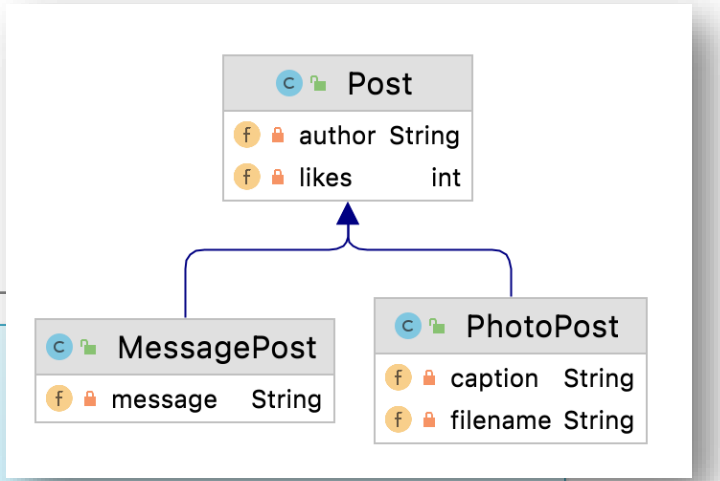
```
0: Leonardo da Vinci
2 people like this.
      Had a great idea this morning.
      But now I forgot what it was. Something to do with flying ...
```

Create this MessagePost

```
1: Alexander Graham Bell
4 people like this.
      experiment.jpg
      I think I might call this thing 'telephone'.
```

Create this PhotoPost

Testing the display method...



```
0: Leonardo da Vinci
2 people like this.
      Had a great idea this morning.
      But now I forgot what it was. Something to do with flying ...

1: Alexander Graham Bell
4 people like this.
      experiment.jpg
      I think I might call this thing 'telephone'.
```

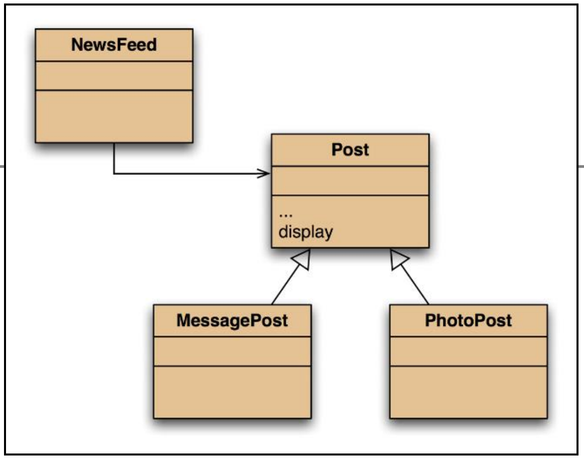
What we want

```
0: Leonardo da Vinci
2 people like this.

1: Alexander Graham Bell
4 people like this.
```

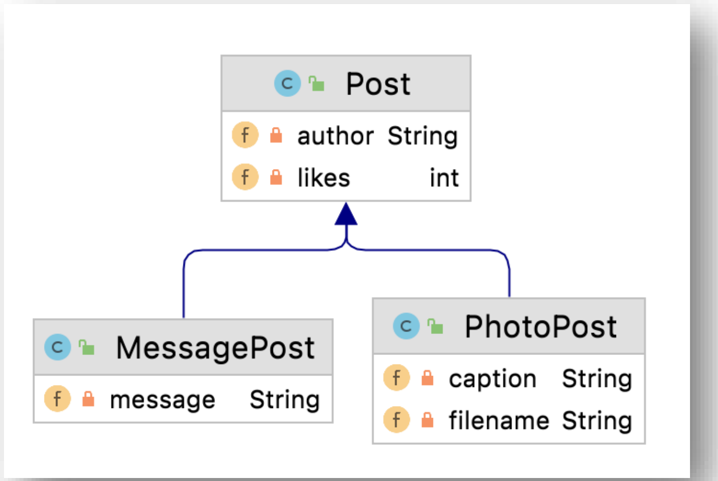
What we have

The problem

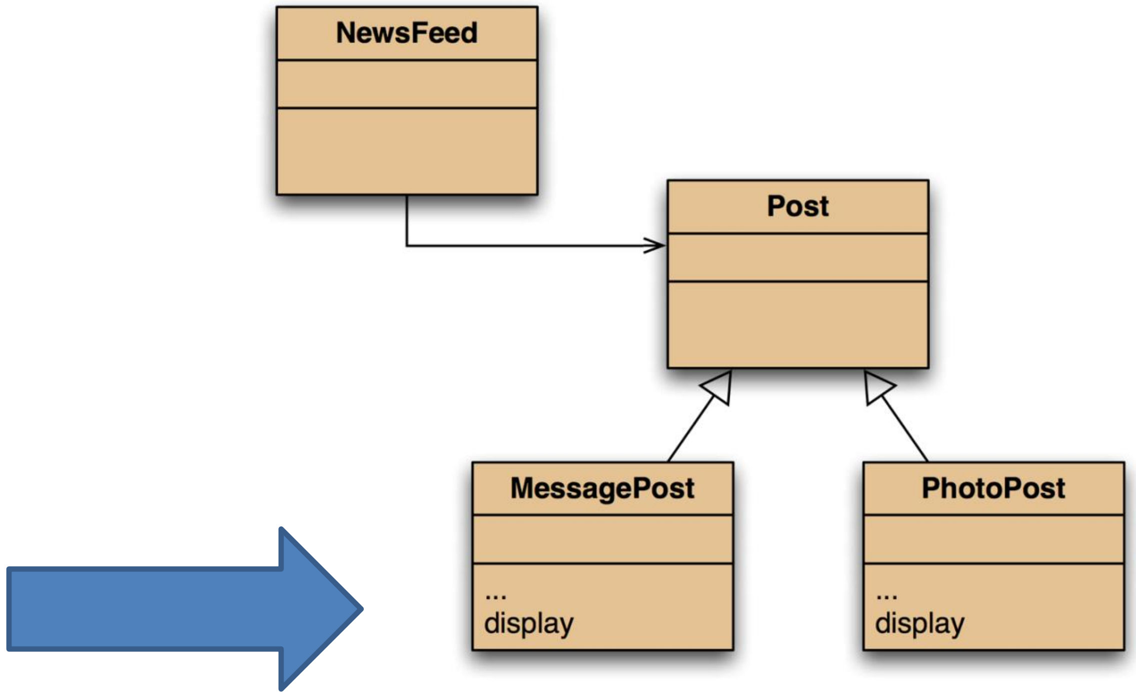


- The **`display`** method in **`Post`** only prints the common fields.

- 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`**.

Topic List

- Method polymorphism
- Static and dynamic type
- Overriding
- Dynamic method lookup
- 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

Static and dynamic type

What is the type of c1?

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

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*.

Static and dynamic type

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

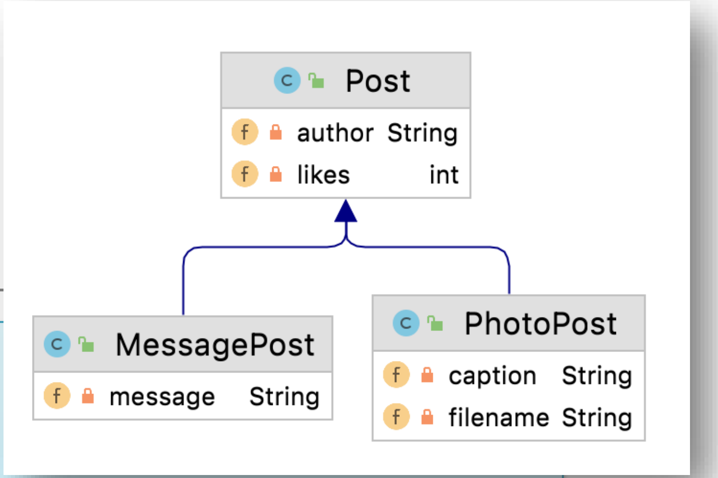
What is the type of v1?

```
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...



```
0: Leonardo da Vinci
2 people like this.
      Had a great idea this morning.
      But now I forgot what it was. Something to do with flying ...

1: Alexander Graham Bell
4 people like this.
      experiment.jpg
      I think I might call this thing 'telephone'.
```

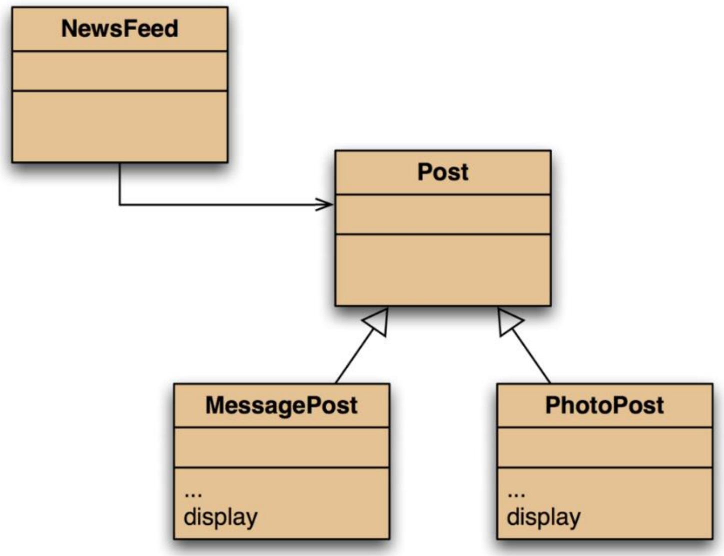
What we *want*

```
0: Leonardo da Vinci
2 people like this.

1: Alexander Graham Bell
4 people like this.
```

What we *have*

Recall our attempt to solve this problem…



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

But:

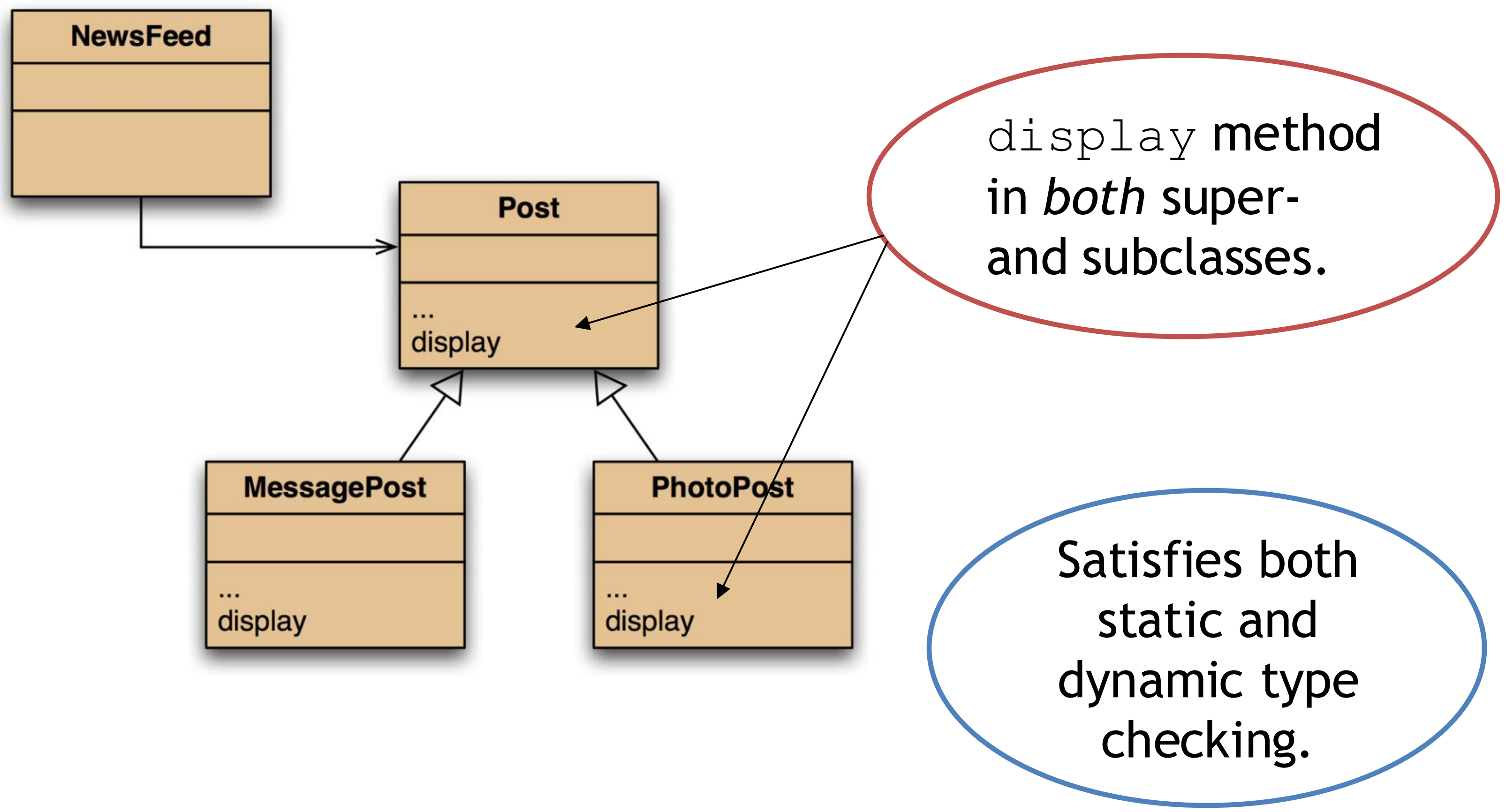
- **`Post`**'s fields are private and **`NewsFeed`** cannot find a **`display`** method in **`Post`**.

```
for(Post post : posts) {
  post.display(); // Compile-time error (static-type violation)
                 // method display() is not found in the
                 // Post class
}
```

Topic List

- Method polymorphism
- Static and dynamic type
- Overriding
- Dynamic method lookup
- Protected access

Overriding - the solution to our problem


NewsFeed
Post
...
display
MessagePost
...
display
PhotoPost
...
display
`display` method in *both* super- and subclasses.
Satisfies both static and dynamic type checking.

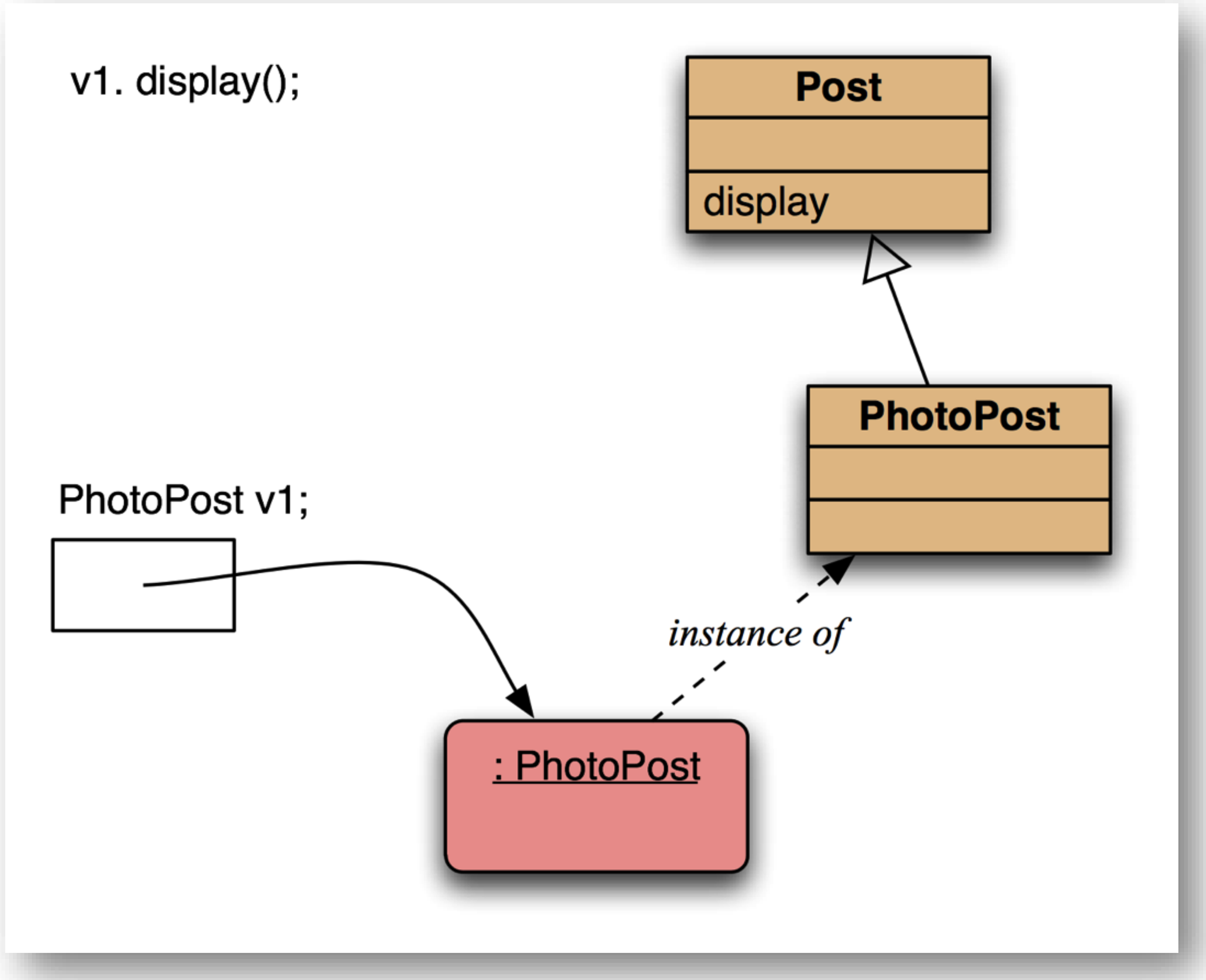
Overriding

- 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?

Topic List

- Method polymorphism
- Static and dynamic type
- Overriding
- Dynamic method lookup
- Protected access

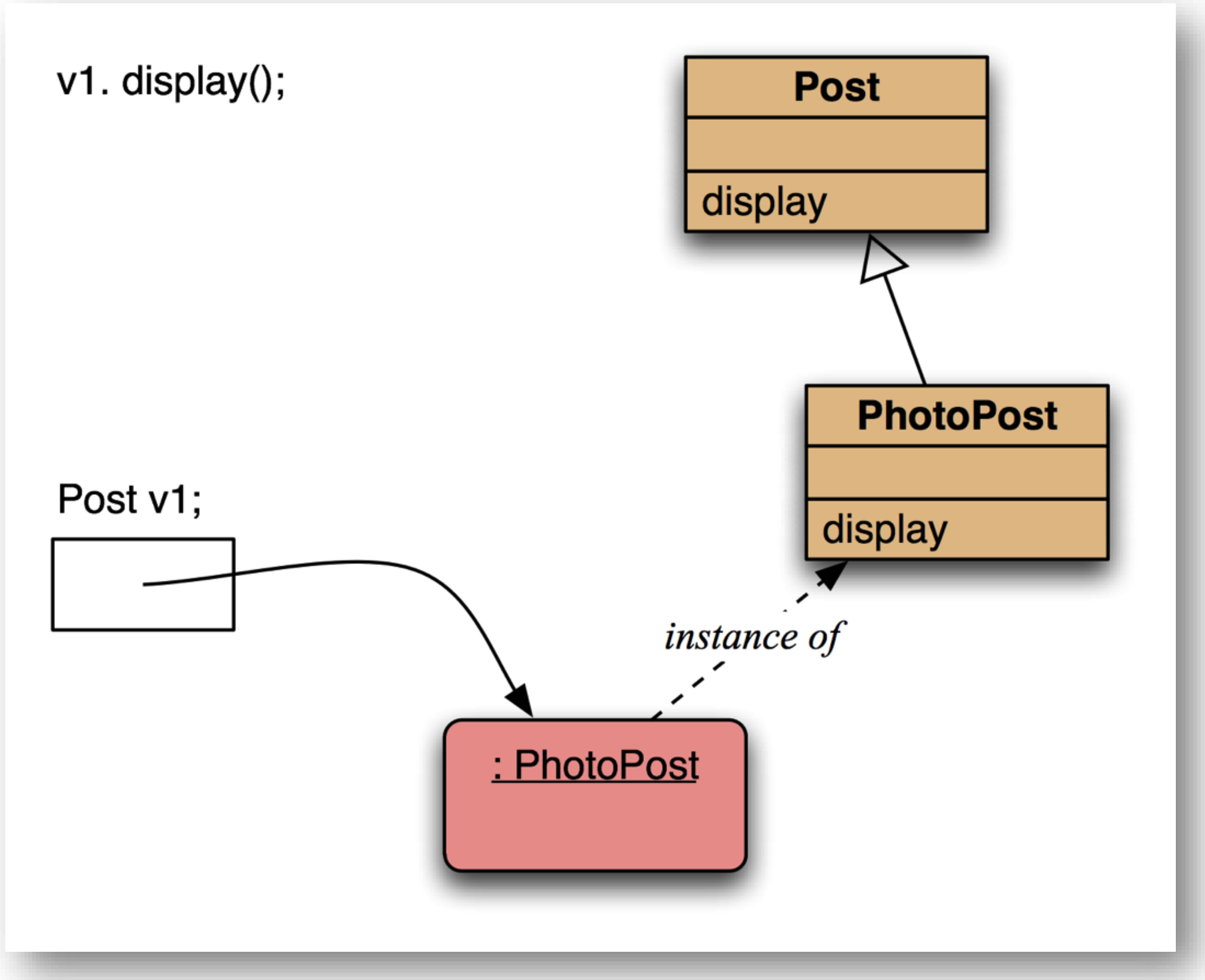
Dynamic method lookup



Inheritance but no overriding.

The inheritance hierarchy is ascended, searching for a match.

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.

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(...)`**
  - Recall we used **`super`** in our constructors.

Calling an overridden method

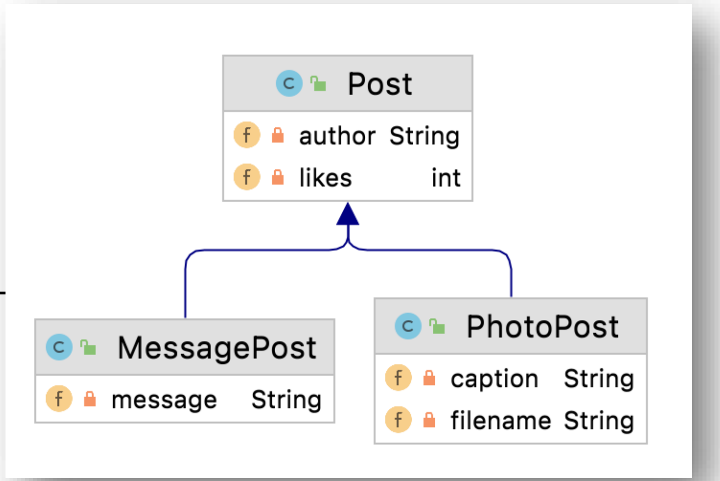
Post

```
public String display() {
    String str = "";

    str += (author + "\n");

    if(likes > 0) {
        str += ("  -  " + likes + " people like this.\n");
    }
    else {
        str += "0 likes.\n";
    }

    return str;
}
```



MessagePost

```
public String display() {
    String str = super.display();

    if (!message.isEmpty()){
        str += "\t" + message + "\n";
    }
    return str;
}
```

Method polymorphism

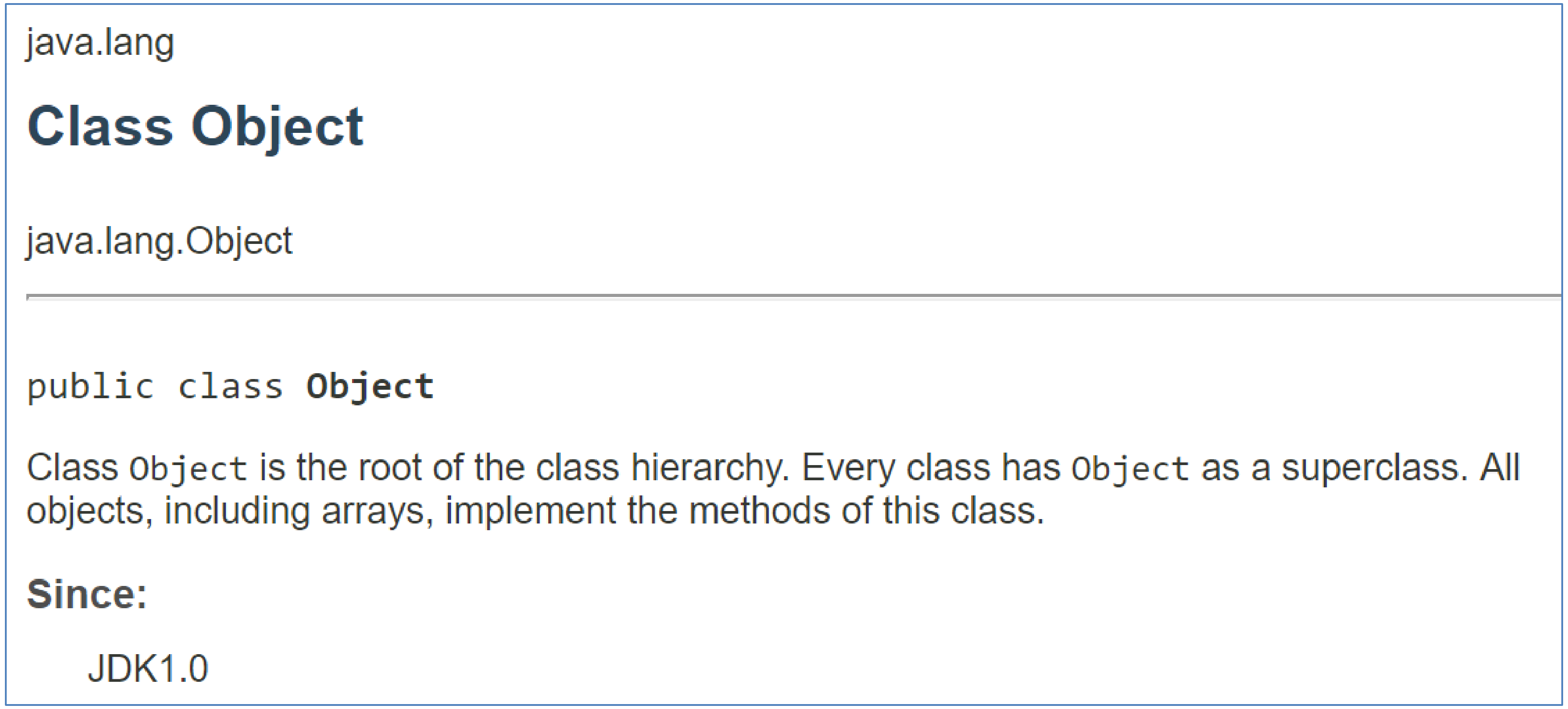
- We have been discussing *polymorphic method dispatch*.
- A polymorphic variable can store objects of varying types.
- Method calls are polymorphic.
  - The actual method called depends on the dynamic object type.

The `instanceof` operator

- Used to determine the dynamic type.
- Can recover ‘lost’ type information.
- 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 object ...
}
```

Recall the Object class...


java.lang

Class Object

java.lang.Object

`public class` **`Object`**

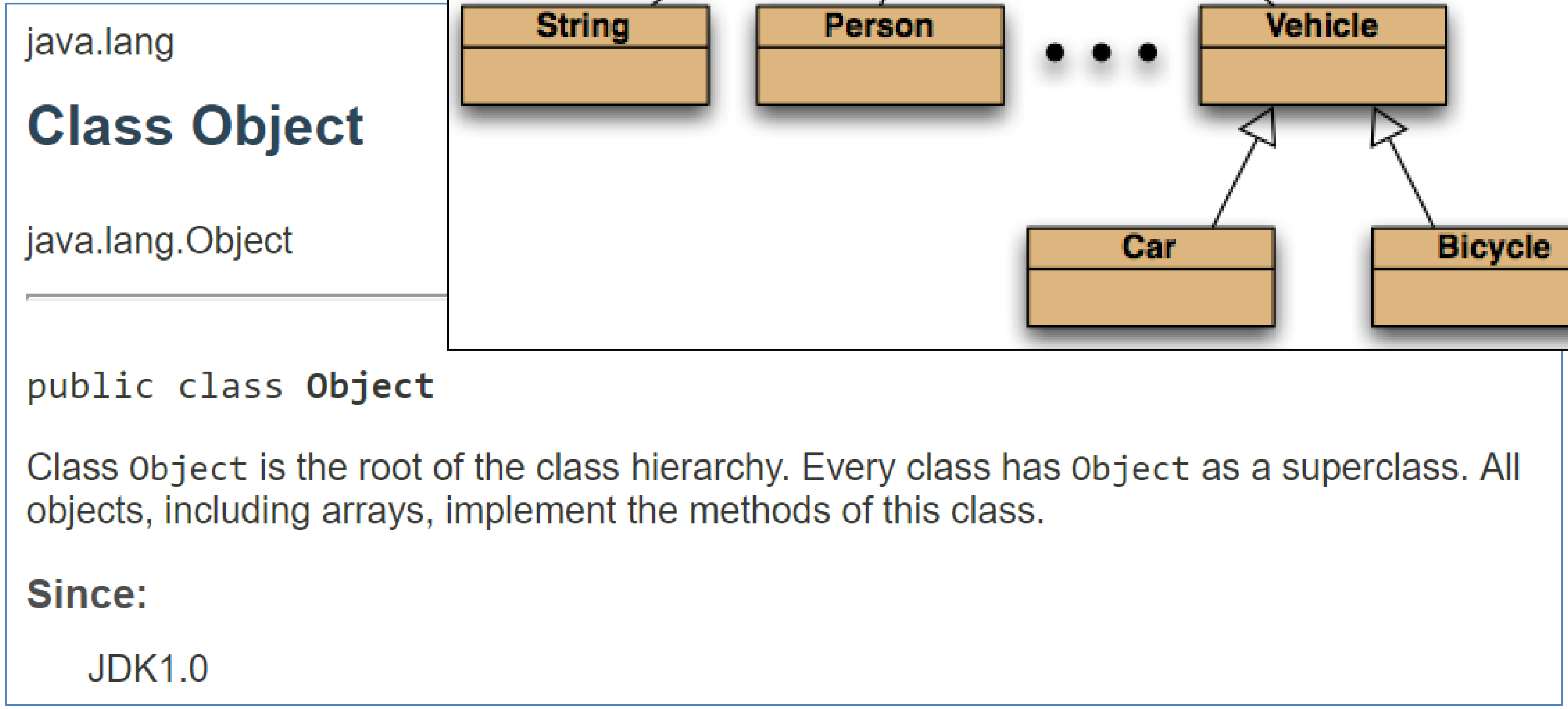
Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.

Since:

JDK1.0

Recall the Object class…

All classes inherit from **Object**.


java.lang

Class Object

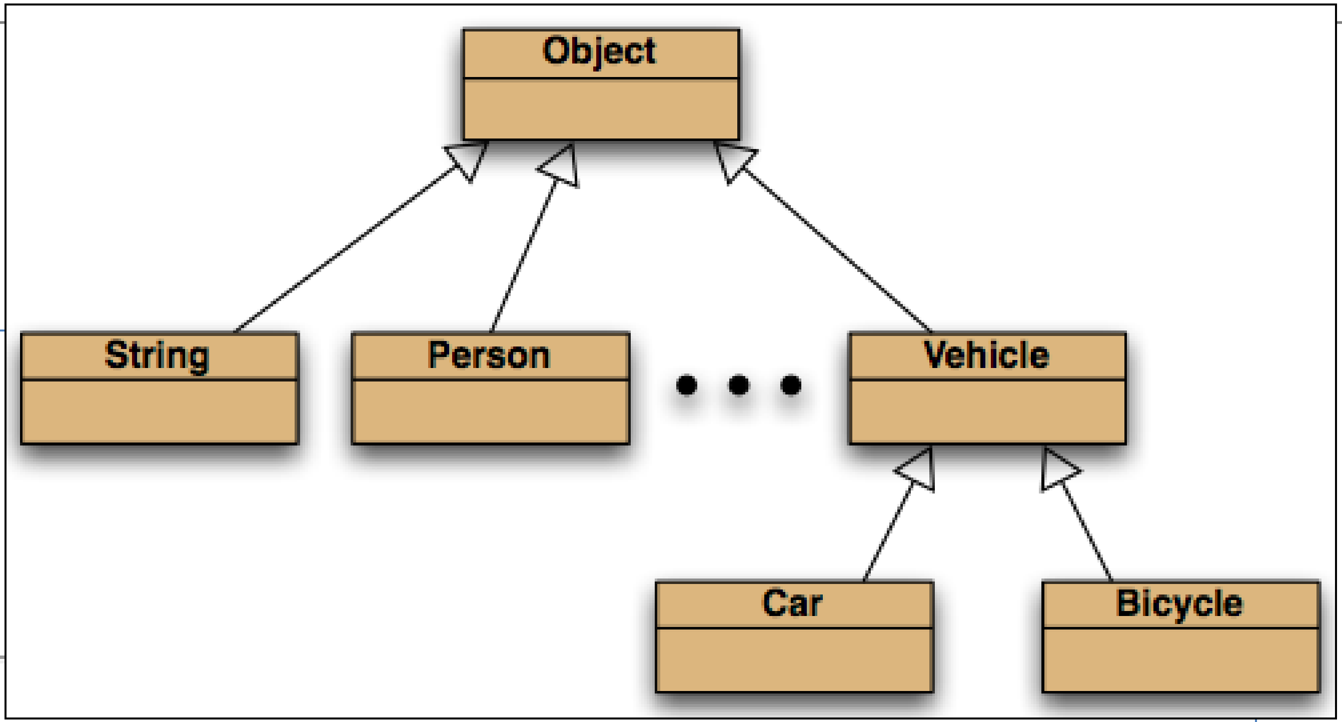
java.lang.Object

```
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.

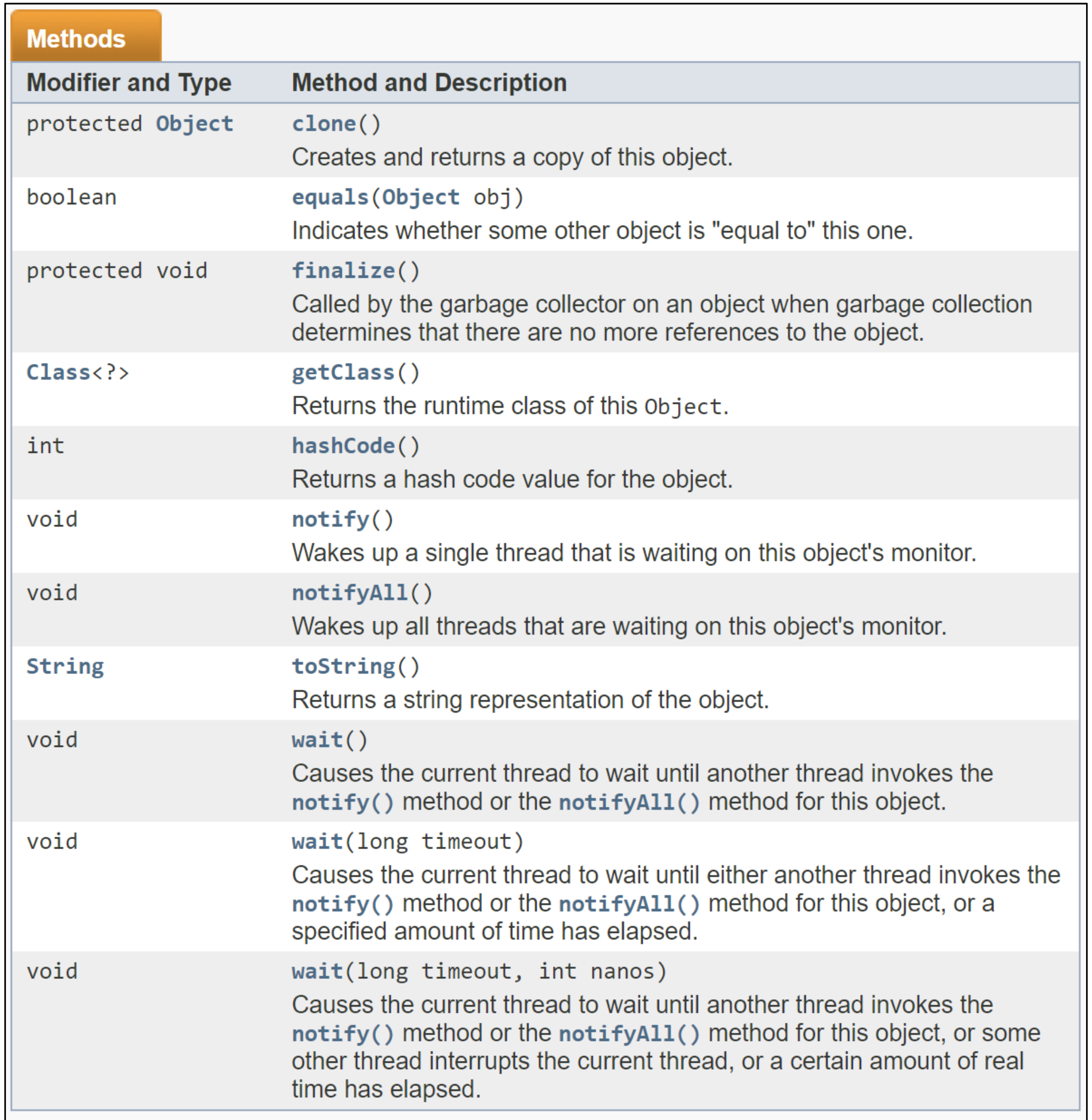
Since:

JDK1.0



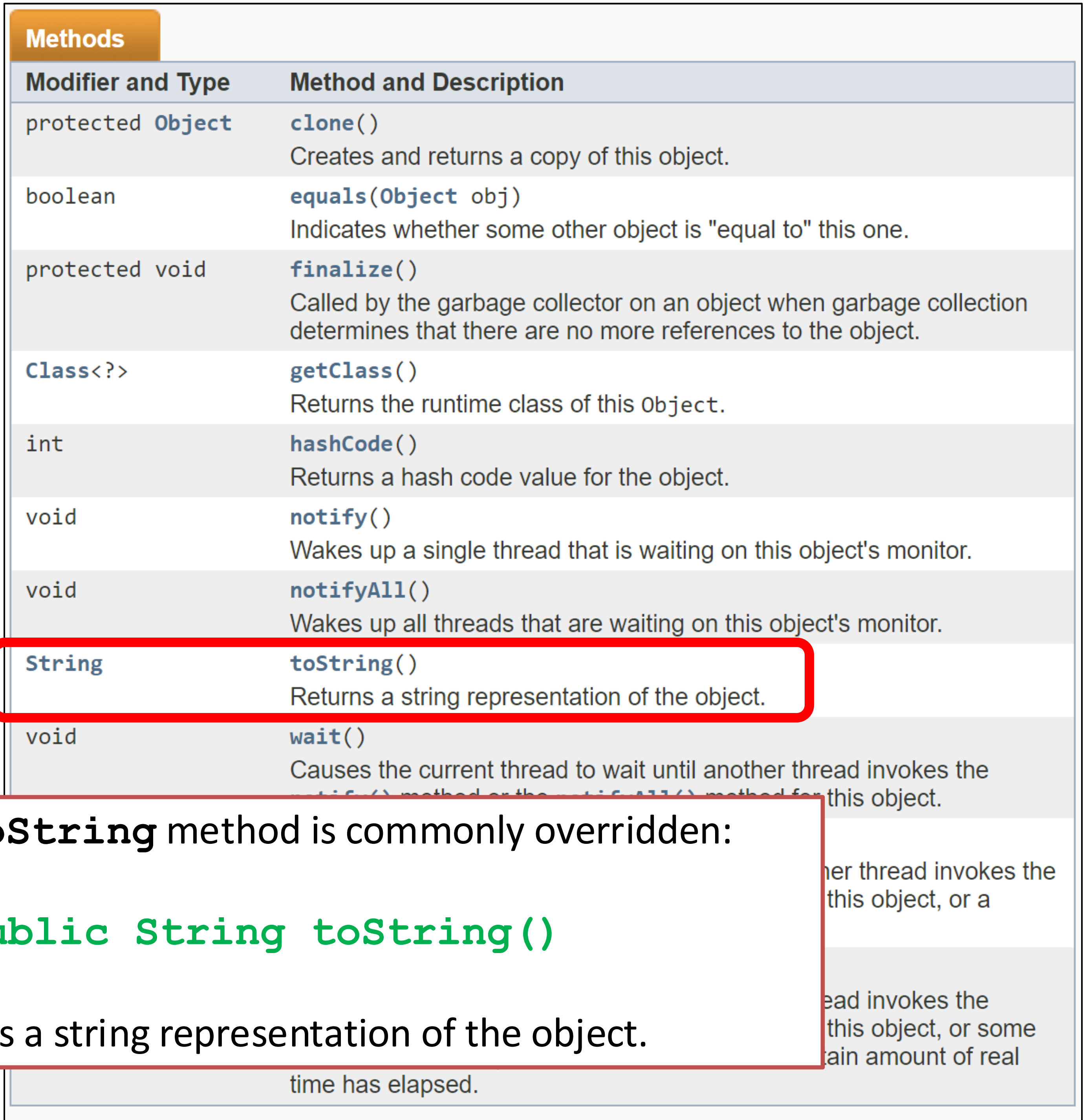
Methods in **`Object`** are inherited by all classes.

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.
	ner 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 = author + "\n";

    if(likes > 0) {
        text += " - " + likes + " people like this.\n";
    }
    else {
        text += "\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);`**

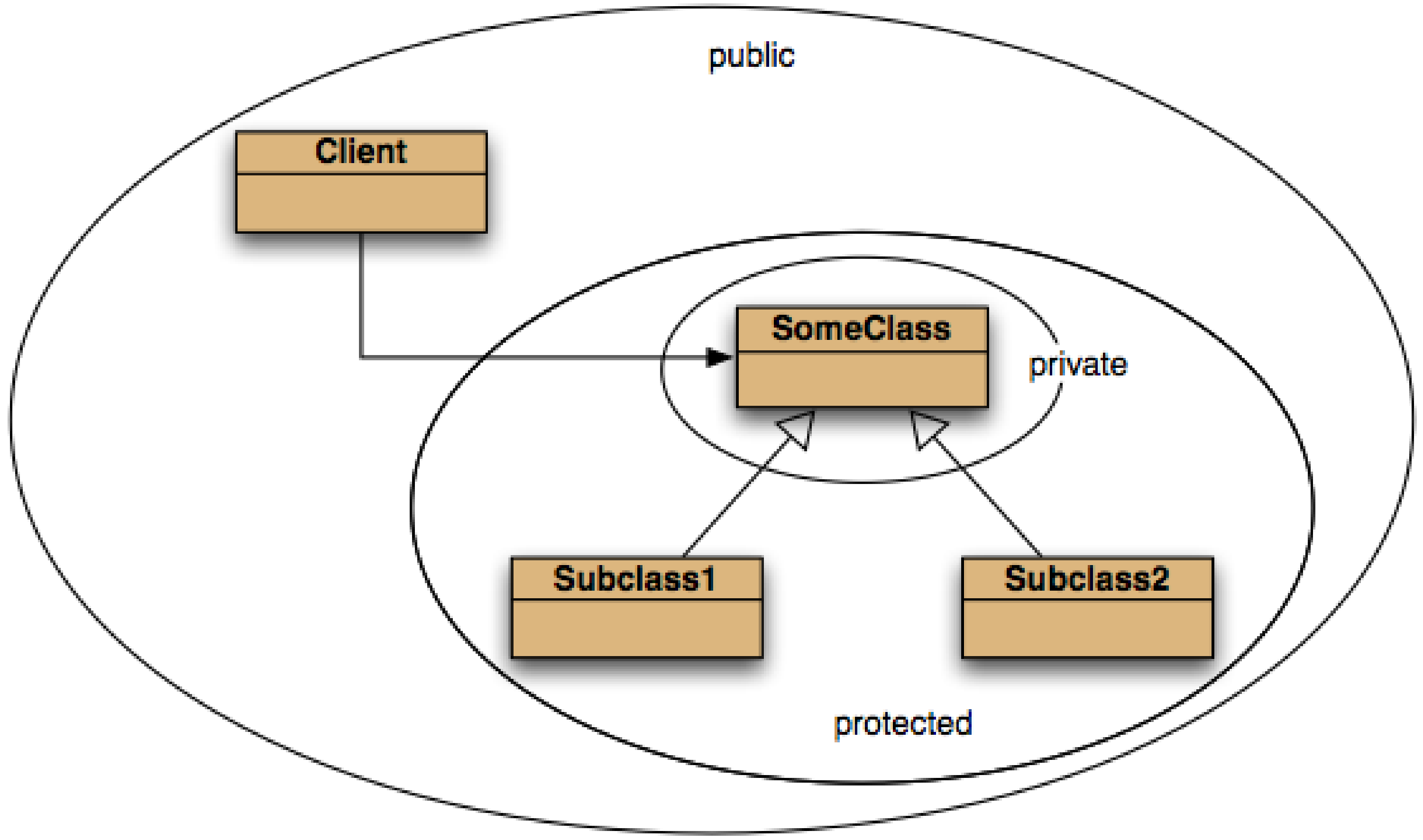
Topic List

- Method polymorphism
- Static and dynamic type
- Overriding
- Dynamic method lookup
- Protected access

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.

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.

Any Questions?