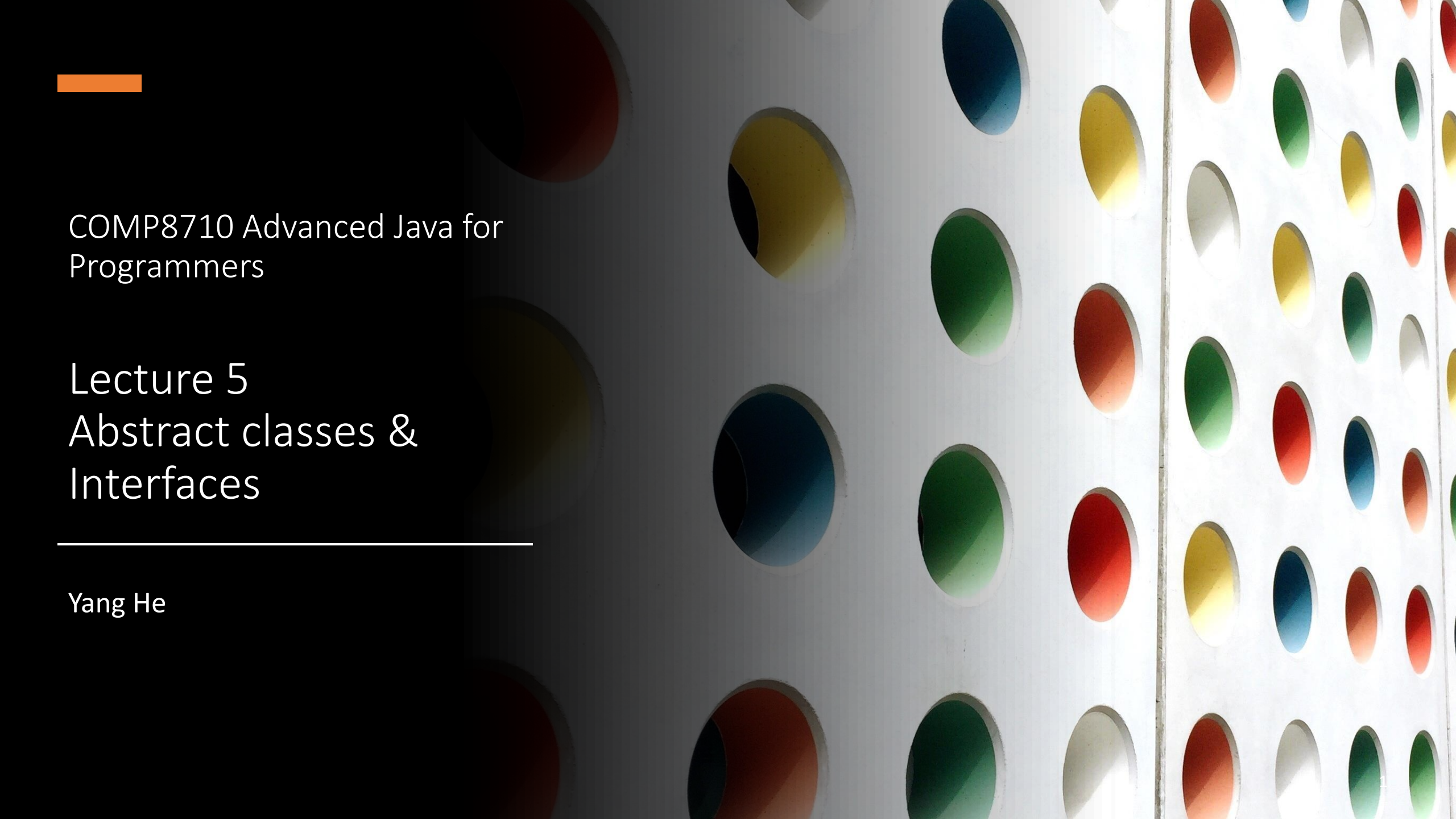
COMP8710 Advanced Java for Programmers

Lecture 5
Abstract classes & Interfaces

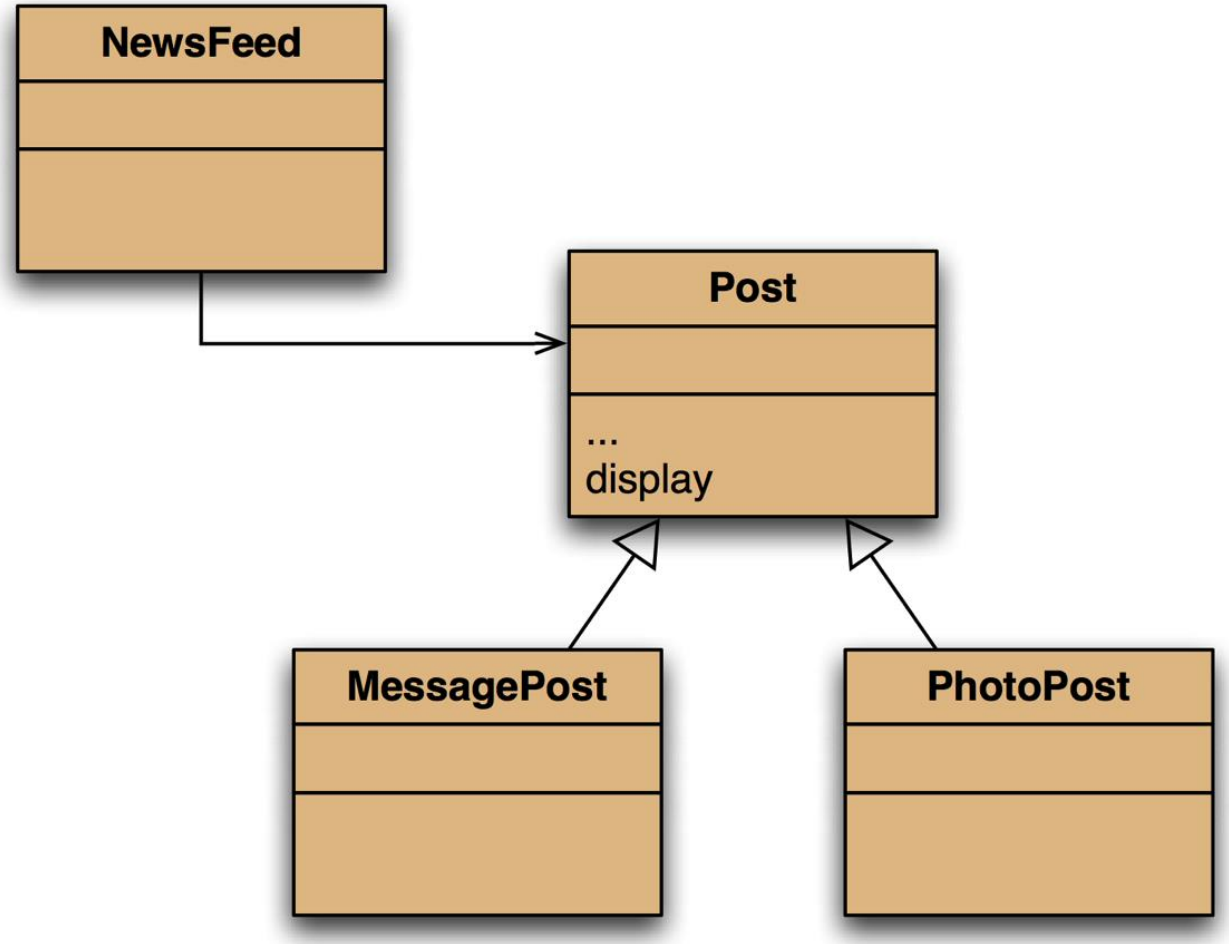
Yang He



Topics

- Static and dynamic type
- Method polymorphism
- Dynamic method lookup
- Method overriding
- Abstract classes
- Interfaces

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.
```

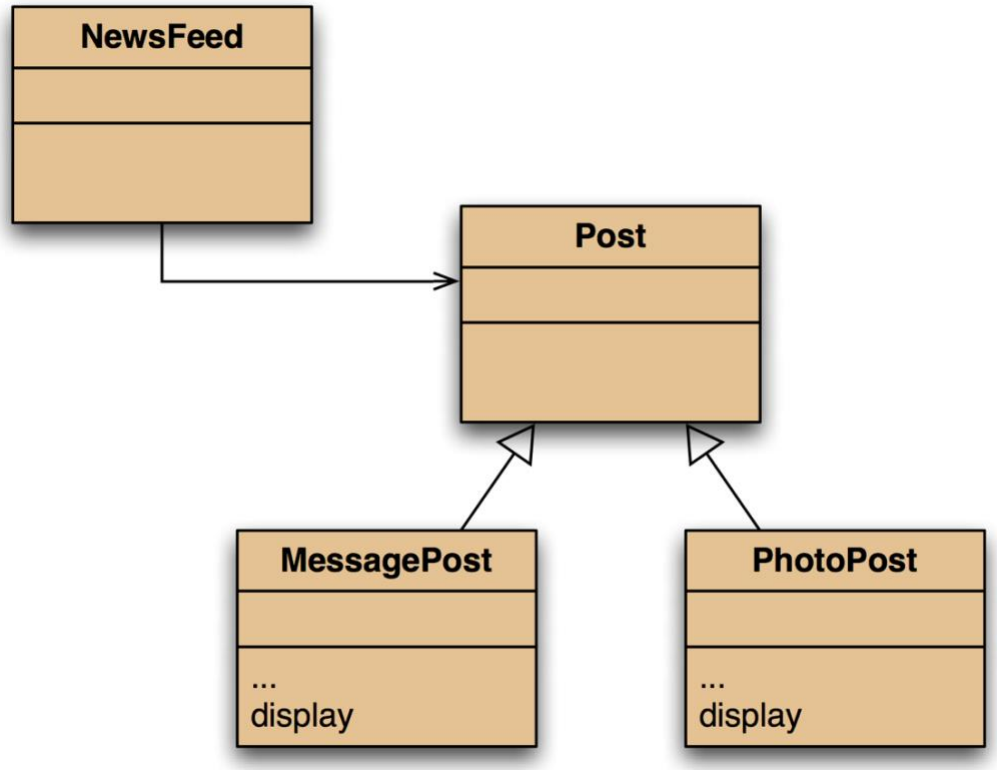

```
Leonardo da Vinci
40 seconds ago  -  2 people like this.
  No comments.

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

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 & dynamic types

Examples

What is the type of c1?

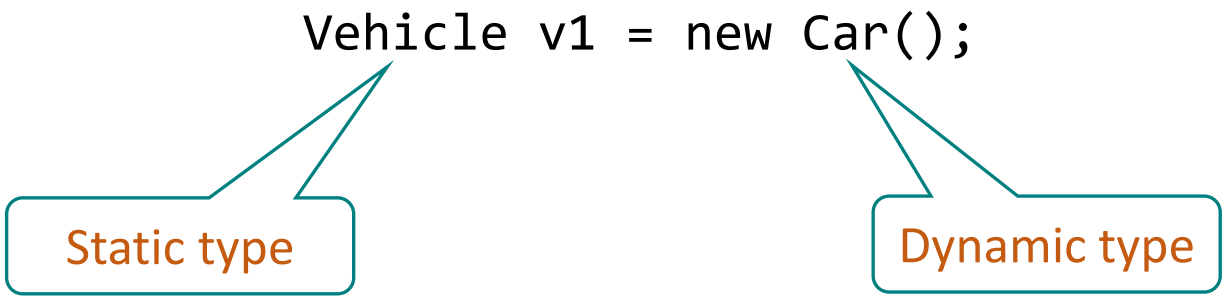
```
Car c1 = new Car();
```

What is the type of v1?

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

Static and dynamic type (1)

- The declared type of a variable is its *static type*
- The type of the object a variable refers to is its *dynamic type*
- E.g.



Static and dynamic type (2)

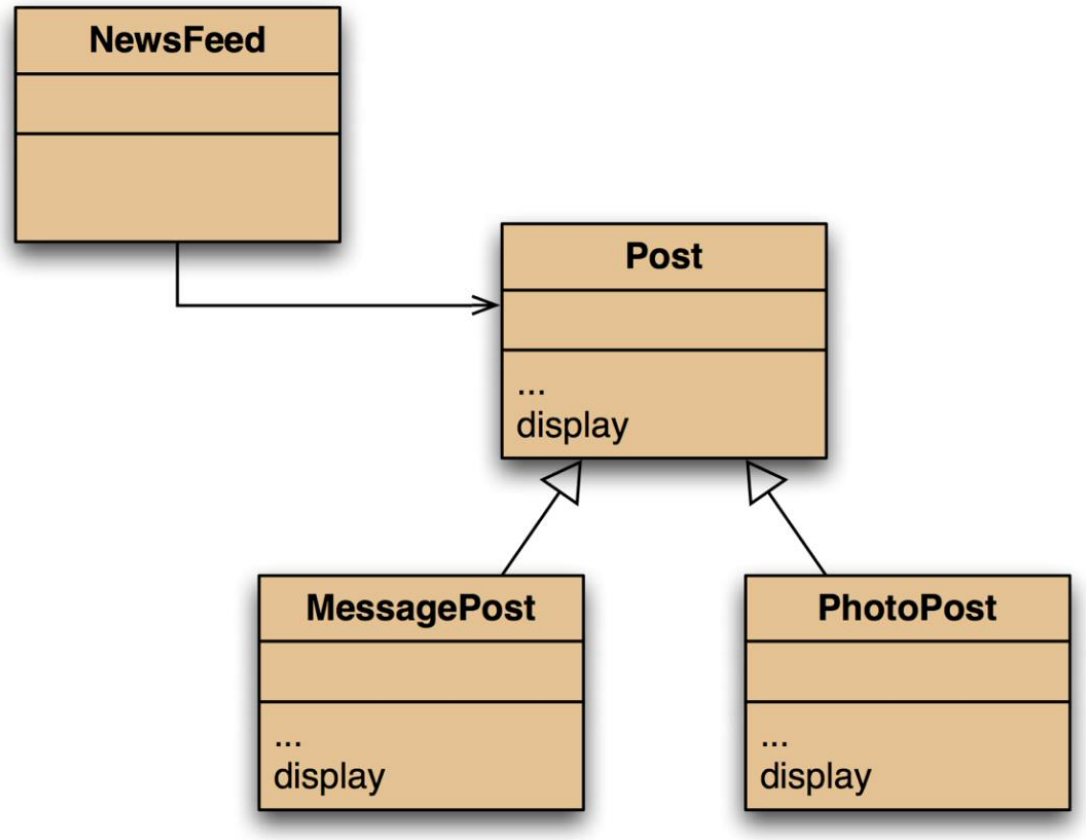
- The compiler's job is to check for static-type violations
- E.g.

```
for(Post post : posts) {
    post.display();     // Compile-time error
}
```

The display method is not defined in the Post class

Overriding: the solution (1)

- The display method in both super- and subclasses
- Satisfies both static and dynamic type checking



Overriding: the solution (2)

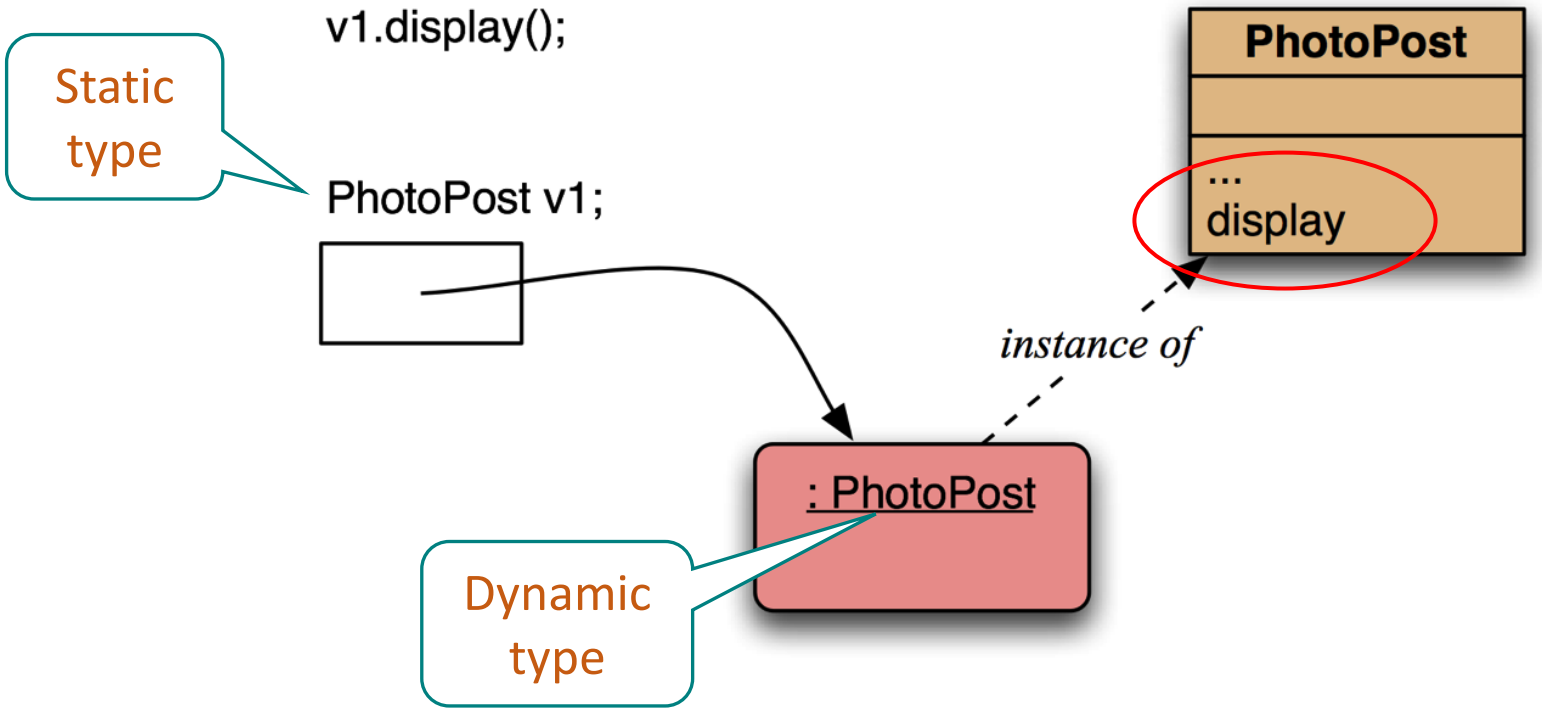
- Superclass and subclass define methods with the same signature
- Each has access to the fields of its class
- Superclass satisfies static type check
- Subclass method is called at runtime – it overrides the superclass version
- What becomes of the superclass version?

Method lookup

Method lookup (1)

Scenario 1:

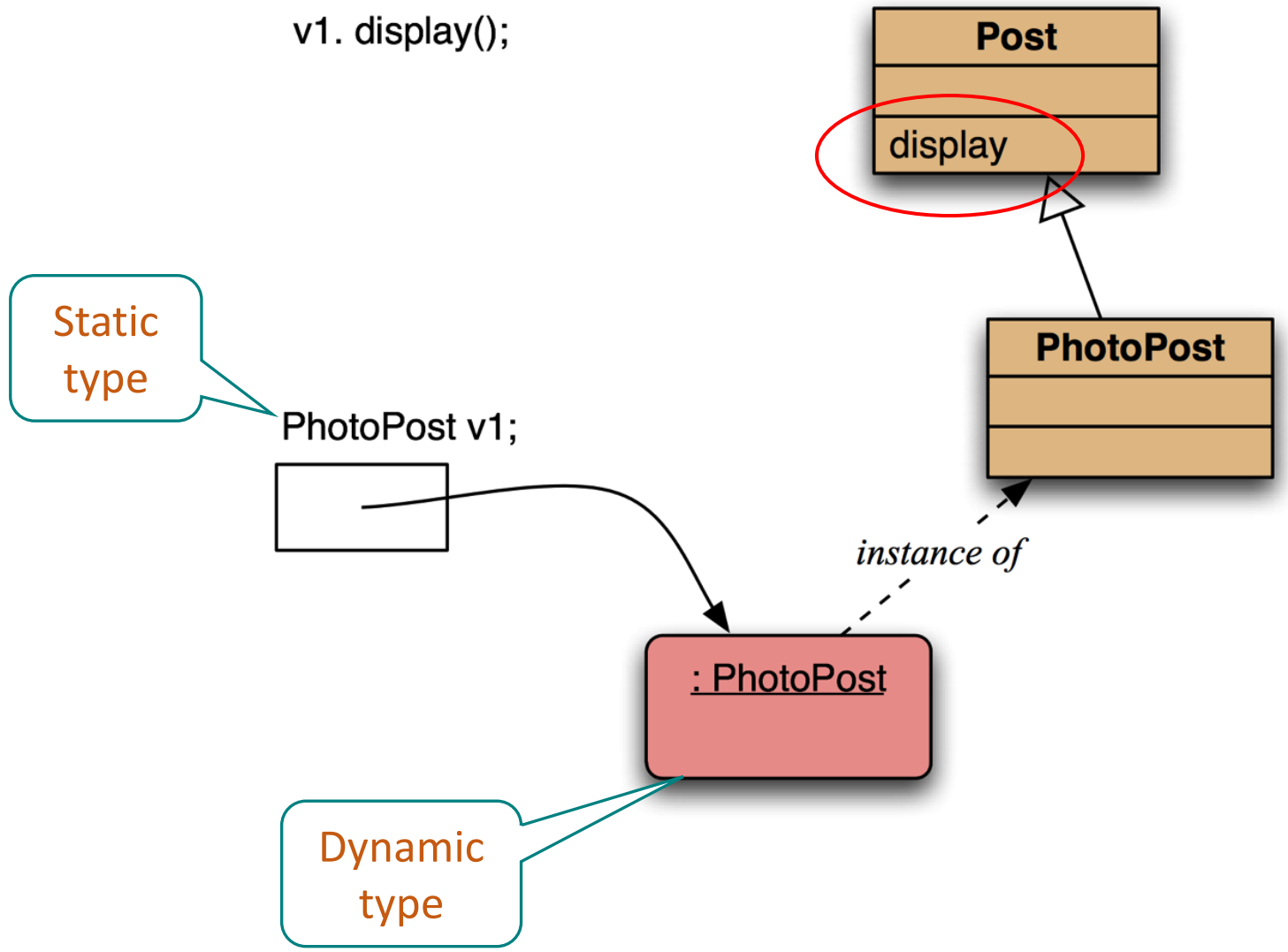
- No inheritance or polymorphism
- The obvious method is selected



Method lookup (2)

Scenario 2:

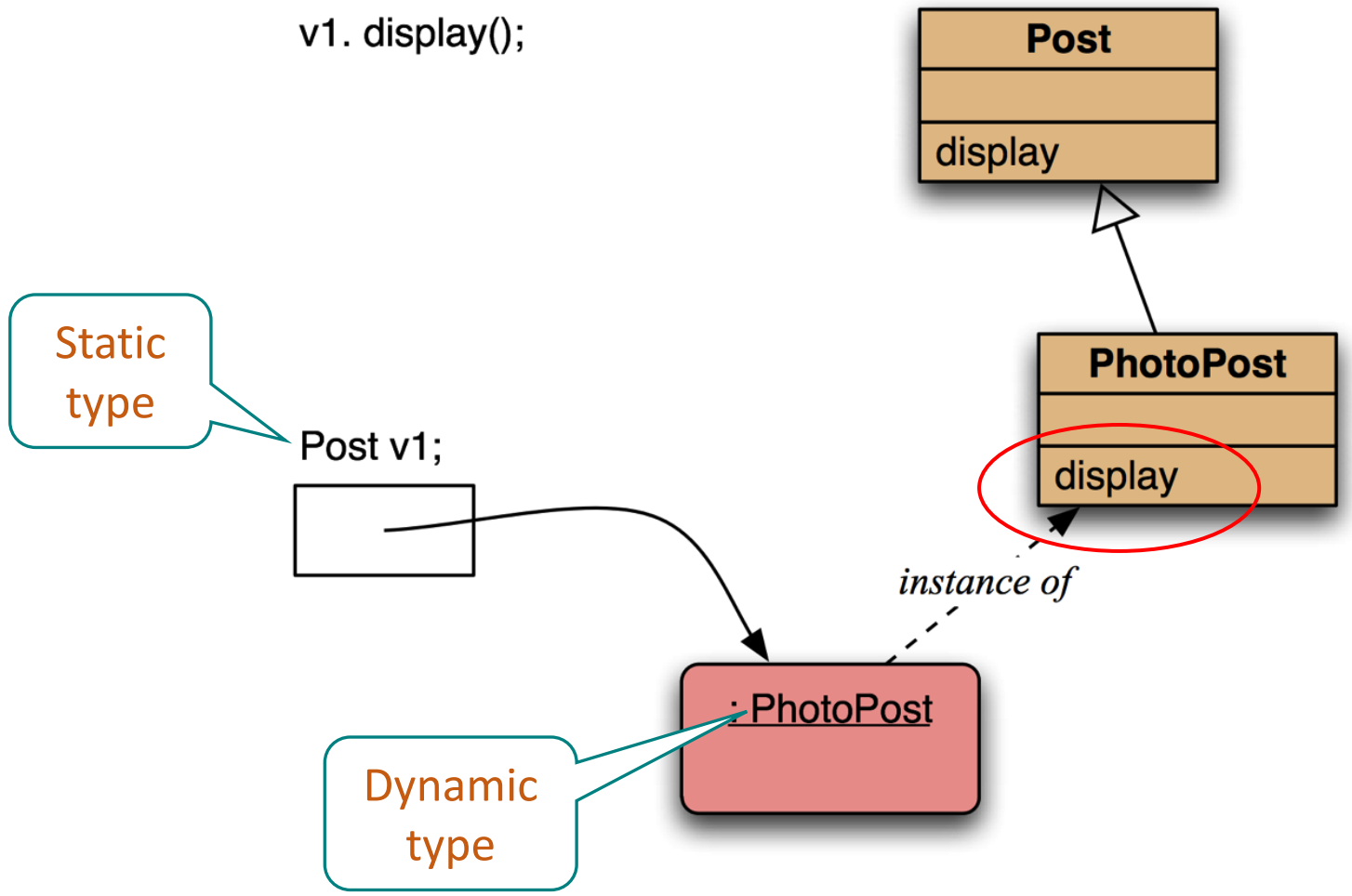
- Inheritance but no overriding
- The inheritance hierarchy is ascended, searching for a match



Method lookup (3)

Scenario 3:

- 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

- E.g. in PhotoPost class:

```
public void display()
{
    super.display();   // call display method of the superclass

    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
  - Identifies ‘lost’ type information
  - Usually precedes assignment with a cast to the dynamic type
- E.g.

```
if (animal instanceof Cat) {
    Cat cat = (Cat) animal;   // casting
    cat.meow();
}
```

- From Java 14, with less boilerplate code:

```
if (animal instanceof Cat cat) {
    cat.meow();
}
```

Overriding methods of Object

- Methods in the Object class are inherited by all classes
- Any of these may be overridden
- The toString and equals methods are commonly overridden
  - toString returns a string representation of the object
  - equals return true if two objects are the same
- println with just an object automatically calls toString

```
System.out.println(post);    // same as System.out.println(post.toString());
```

Overriding equals method

- E.g. Overriding equals in Book

```
public boolean equals(Object obj)
{
   // check if they refer to the same object
   if (this == obj) return true;

   // check if obj is an object of Book
   if (!(obj instanceof Book other)) return false;

   return this.title.equals(other.title) &&
          this.author.equals(other.author)
}
```

Quiz

- Assume the code below compiles successfully.

```
Pet pet = new Cat();
pet.sleep();
```

a) What is the static type of pet?

b) What is the dynamic type of pet?

c) Which class must have sleep method: Pet or Cat?

d) If both Pet and Cat have the sleep method, which one is called?

Abstract classes

An example

- E.g. a list of Animal objects

```
var animals = new ArrayList<Animal>();
```

- To print out the sounds made by all animals in the list:

```
for (Animal a: animals) {
     System.out.println(a.sounds());
}
```

- *Animal is a superclass*
- *Examples of the subclasses: Cat, Dog, etc.*

- Static type checking requires a sounds() method in the Animal class
- However, there is no obvious shared implementation among different animals

Abstract method

- We can define the sounds() method as an abstract method
- An abstract method has *no* body, its signature is followed by a ;
- The abstract keyword indicates that this is intentional – we require all subclasses to override this method
- E.g. the subclass Cat is concrete enough for sounds() to be implemented

```
abstract class Animal {
  ...
  public abstract String sounds();
  ...
}

class Cat extends Animal {
  ...
  @Override
  public String sounds() {
     return "Meow";
  }
  ...
}
```

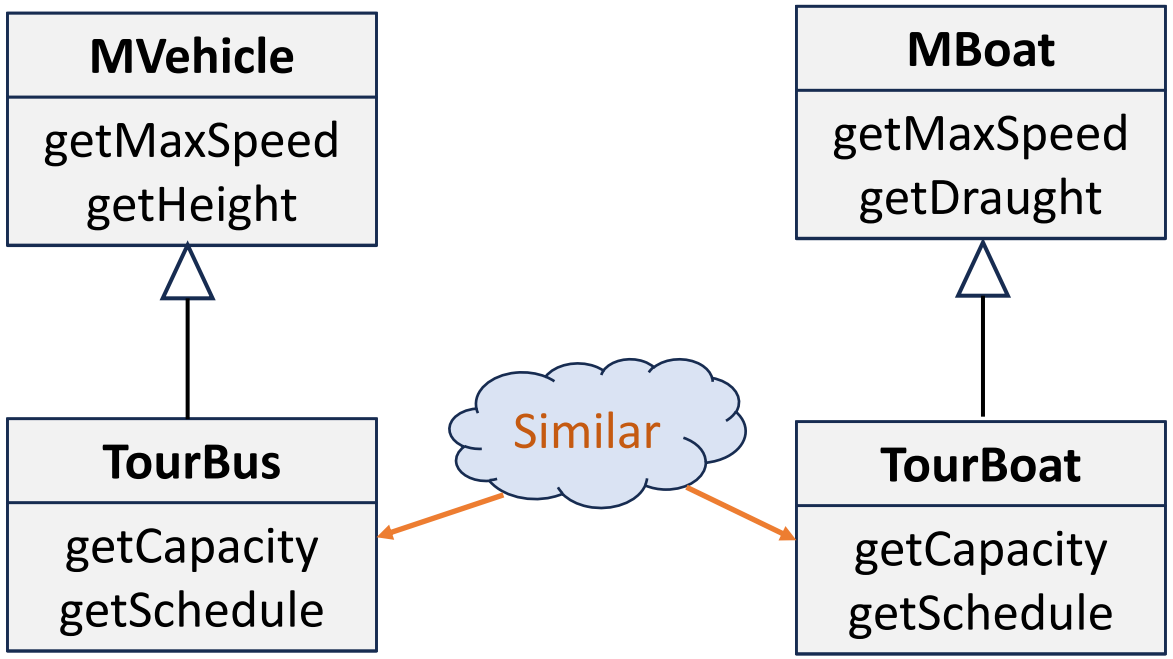
Abstract classes

- A class with an abstract method must be an abstract class
- Abstract classes cannot be instantiated
- An abstract class does not have to have an abstract method!
- Concrete subclasses must complete the implementation of the abstract methods

```
public abstract class Animal {
    ...
    public abstract String sounds();
    ...
}
```

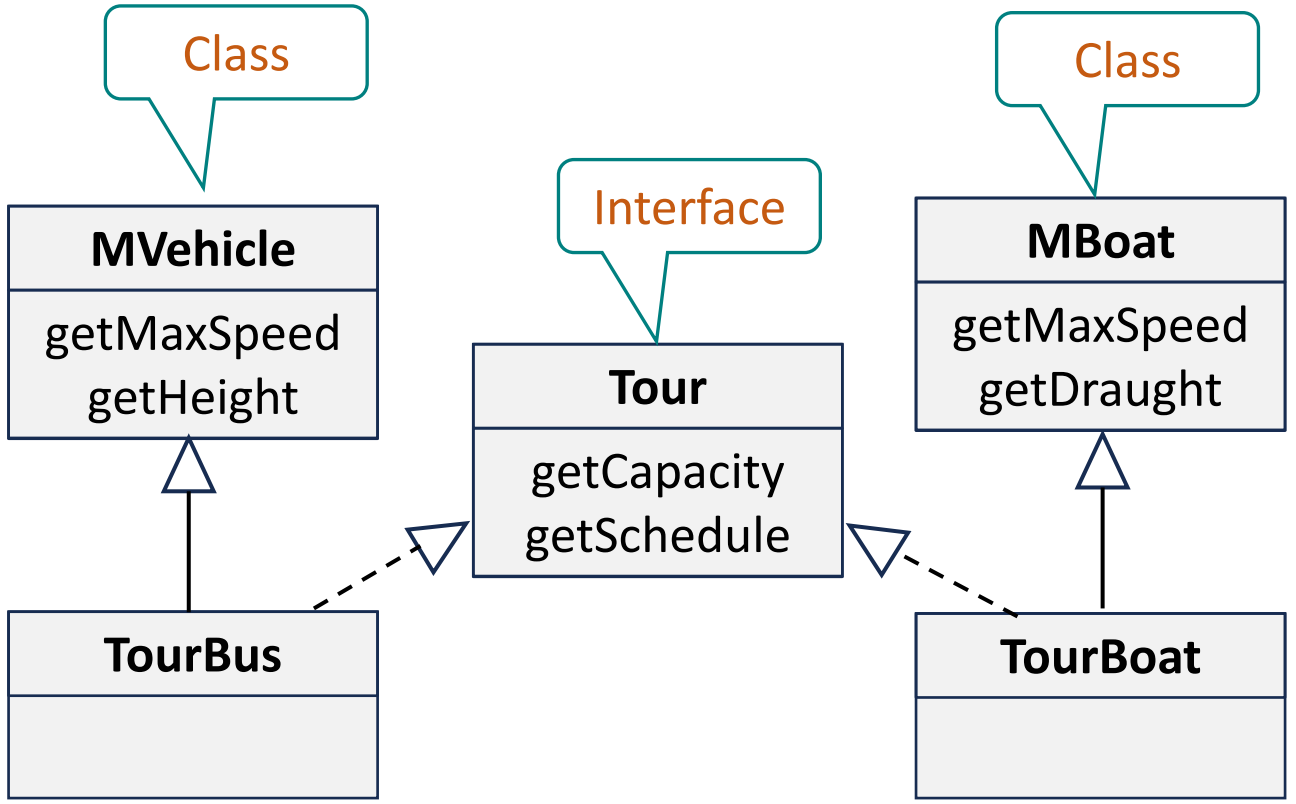
Designing classes

- E.g.



Multiple inheritance

- Having a class inherit directly from multiple ancestors
- Each language has its own rules
- Java forbids it for classes
- Java permits it for interfaces



Interfaces (1)

- An interface is a class skeleton (like an abstract class) that defines method signatures (and static members)
- E.g.

```
public interface Tour {
    int getCapacity();
    Schedule getSchedule(Date date);
}
```

- The methods within the interface are public and abstract (*the keywords are usually omitted)*

Classes implement an interface

- A class can implement one or more interfaces by providing methods for their signatures
- E.g.


```
public class TourBus extends MVehicle implements Tour
{
    ...

    public int getCapacity() { // code for this method }

    public Schedule  getSchedule(Date  date) { // code for this method}

    ...
}
```

Interfaces features

- Interfaces do not define constructors
  - They cannot be instantiated
- All methods are public
- All fields are public, static and final
- Abstract methods may omit the keyword abstract
- From Java 8: Methods marked as default or static have a method body – they are concrete methods, not abstract
- From Java 9: we can define private concrete methods in an interface

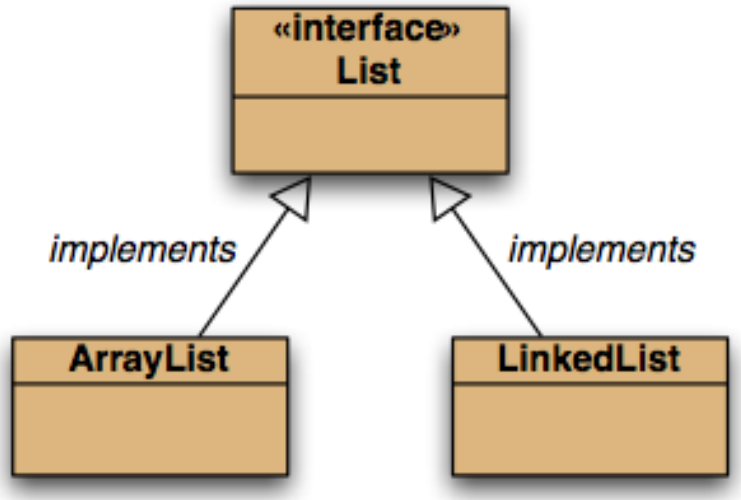
Interfaces as types

- Implementing classes are subtypes of the interface type
- So, polymorphism is available with interfaces as well as classes
- E.g.

```
Tour tour1 = new TourBus(…);

tour1.getCapacity()
```

Interfaces as specifications

- Strong separation of functionality from implementation
  - Though parameter and return types are mandated
- Clients interact independently of the implementation
  - But clients can choose from alternative implementations
  - List, Map and Set are examples



Empty interfaces

- Occasionally, an interface is defined containing neither methods nor fields
- These empty interfaces are called marker interfaces
- A class that implements a marker interface adds nothing to its functionality
- The interface flags that the class satisfies some property
  - E.g. Cloneable – if a programmer wishes to make it possible for an object to duplicate (clone) itself, then the class of the object must implement Cloneable