Inheritance

SOftEng
http://softeng.polito.it

Inheritance

- A class can be a sub-type of another class
- The derived class
  - Implicitly contains (inherits) all the members of the class it inherits from
  - Plus, it can augment its structure with any additional member that it defines explicitly
  - Can also override the definition of existing methods by providing its own implementation
- The code of the derived class consists only of changes and additions to the base class

Addition

```
class Employee{
   String name;
   double wage;
   void incrementWage(){…}
}
```

```
class Manager extends Employee{
   String managedUnit;
   void changeUnit(){…}
}
```

```
Manager m = new Manager();
m.incrementWage(); // OK, inherited
```

Overriding

```
class Vector{
   int vect[];
   void add(int x) {…}
}
```

```
class OrderedVector extends Vector{
   void add(int x){…}
}
```

Inheritance and polymorphism

```
class Employee{
 private String name;
 public void print(){
  System.out.println(name);
 }
}
class Manager extends Employee{
  private String managedUnit;
  public void print(){ //overrides that in Employee
   System.out.println(name); //un-optimized!
   System.out.println(managedUnit);
  }
}
```

Inheritance and polymorphism

```
Employee e1 = new Employee();
Employee e2 = new Manager();
e1.print();
e2.print();
```

Correct: a **Manager** is an **Employee**

Employee version: prints only the name

Manager version: prints name and unit

SOftEng
http://softeng.polito.it

Why inheritance – Reuse

- Frequently, a class is merely a modification of another class: in this way, inheritance helps minimize repetition of the same code
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionality in the base class automatically adds it in the subclasses too
  - Less chances of different (and inconsistent) implementations of the same operation

Why inheritance – Flexibility

- Often, one needs to treat objects from different classes in a similar way
  - Polymorphism: allows feeding algorithms with different objects
  - Dynamic binding: allows accommodating different behavior behind the same interface

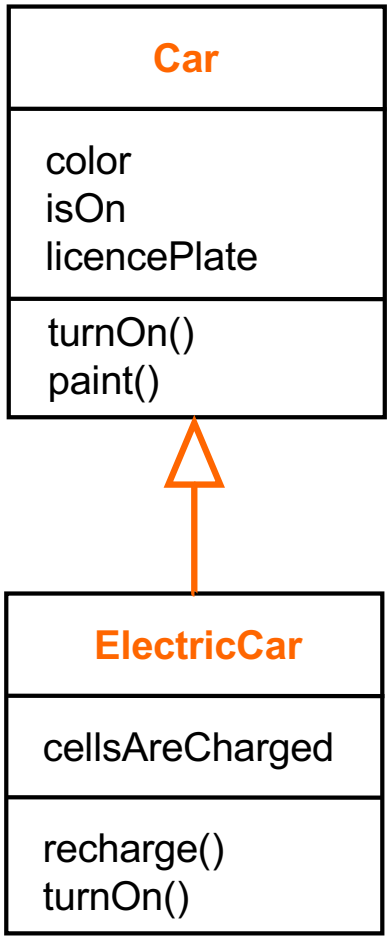
Inheritance in real life

- A new design created by the modification of an already existing design
  - The new design consists of only the changes or additions from the base design
- CoolPhoneBook inherits from PhoneBook
  - E.g., adds mail address and cell number

Inheritance in few words

- Subclass
  - Inherits attributes and methods defined in the base classes
  - Can modify inherited attributes and methods (override)
  - Can add new attributes and methods

Inheritance in Java: extends




```
class Car {

  String color;
  boolean isOn;
  String licencePlate;

  void paint(String color) {
    this.color = color;
  }

  void turnOn() {
    isOn=true;
  }
}
```

```
class ElectricCar extends Car
{

  boolean cellsAreCharged;

  void recharge() {
    cellsAreCharged = true;
  }

  void turnOn() {
    if(cellsAreCharged )
      isOn=true;
  }
}
```

ElectricCar

- Inherits
  - Attributes: **`color`**, **`isOn`**, **`licencePlate`**
  - Methods: **`paint()`**
- Modifies (overrides)
  - Methods: **`turnOn()`**
- Adds
  - Attributes: **`cellsAreCharged`**
  - Methods: **`recharge()`**

Multi-level inheritance

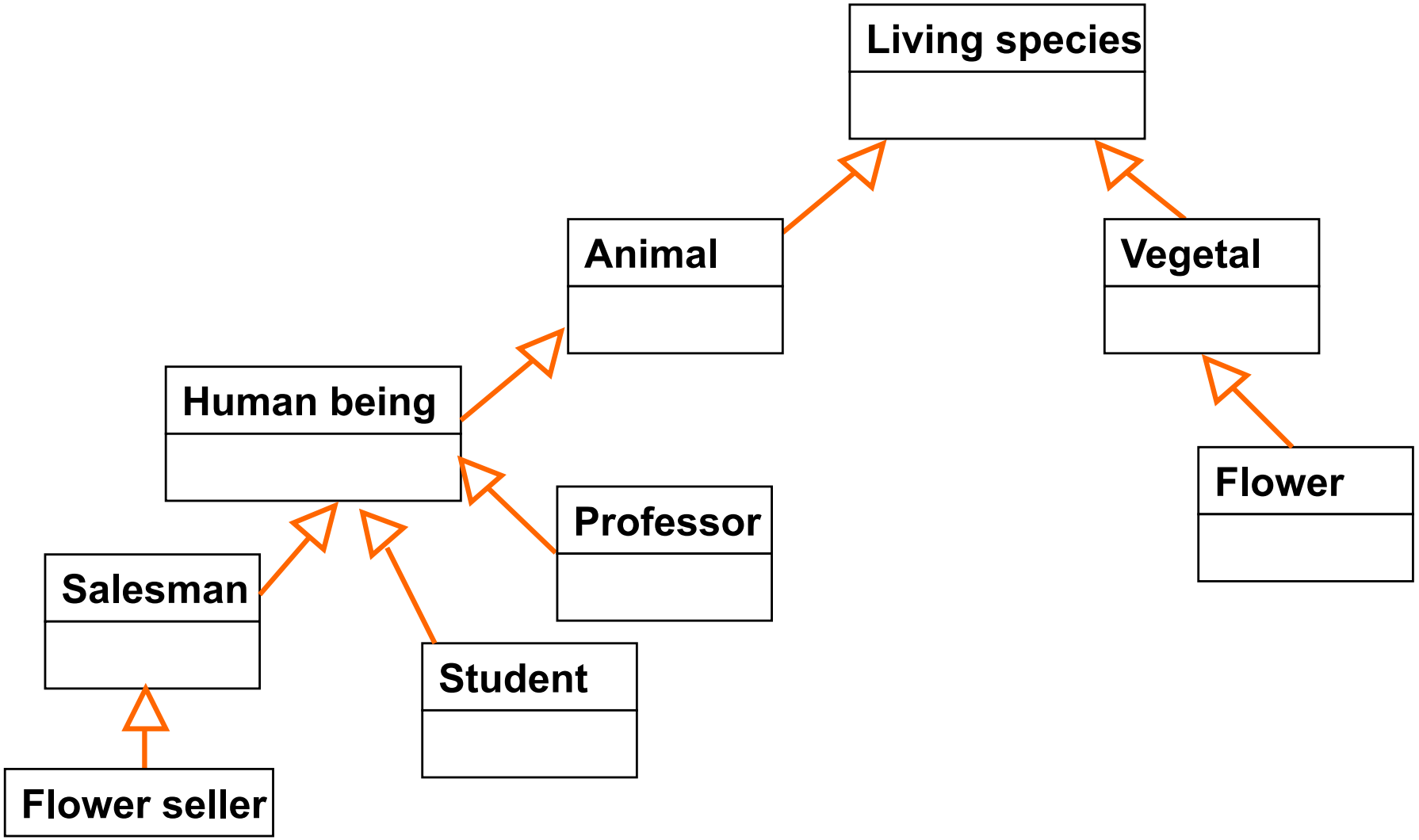
- Inheritance can be applied at multiple stages

```
class B extends A { ... }
```

```
class C extends B { ... }
```

```
class D extends C { ... }
```

Example of inheritance tree



Inheritance terminology

- Class one above
  - Parent class
- Class one below
  - Child class
- Class one or more above
  - Superclass, ancestor class, base class
- Class one or more below
  - Subclass, descendent class

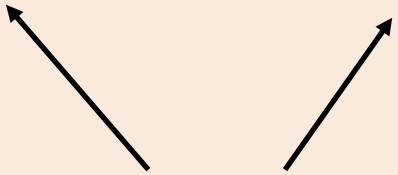
Visibility (scope)

SOftEng
http://softeng.polito.it

Example

```
class Employee {
  private String name;
  private double wage;
}
```

```
class Manager extends Employee {

  void print() {
    System.out.println("Manager" +
                       name + " " + wage);
  }
}
```



Not visible

Protected

- Attributes and methods marked as
  - `public` are always accessible
  - `private` are accessible within the declaring class only
  - `protected` are accessible within the class and its subclasses

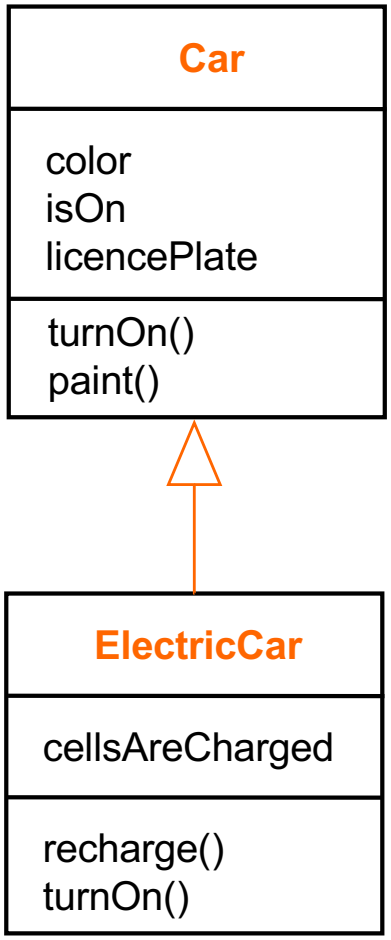
In summary

	Method in the same class	Method of another class in the same package	Method of subclass	Method of class in another package
private	✓			
package	✓	✓		
protected	✓	✓	✓	
public	✓	✓	✓	✓

Super (reference)

- `this` is a reference to the current object
- `super` is a reference to the parent class

Example




```
class Car {

  String color;
  boolean isOn;
  String licencePlate;

  void paint(String color) {
    this.color = color;
  }

  void turnOn() {
    isOn=true;
  }
}
```

was

```
if(cellsAreCharged)
  isOn = true;
```

```
class ElectricCar extends Car{

  boolean cellsAreCharged;

  void recharge() {
    cellsAreCharged = true;
  }

  void turnOn() {
    if( cellsAreCharged )
      super.turnOn();
  }
}
```

Inheritance and constructors

SOftEng
http://softeng.polito.it

Construction of child objects

- Since each object "contains" an instance of the parent class, the latter must be initialized
- Java compiler automatically inserts a call to default constructor (no params) of parent class, **`super()`**
- The call is inserted as the first statement of each child constructor

Construction of child objects

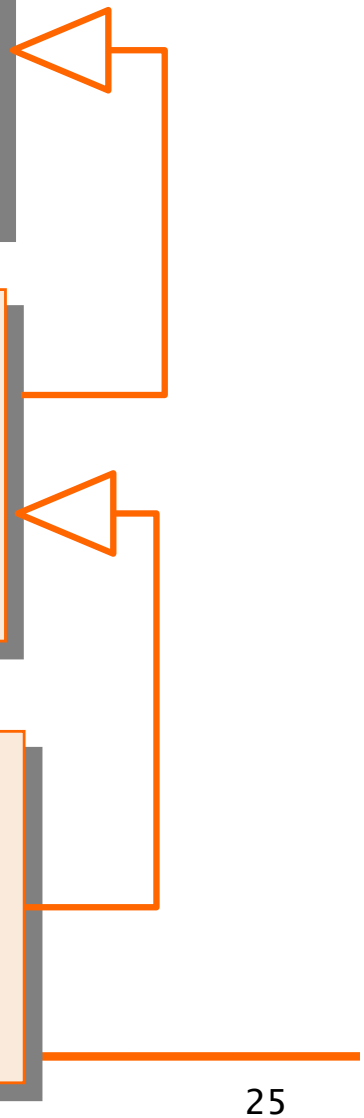
- Execution of constructors proceeds top-down in the inheritance hierarchy
- In this way, when a method of the child class is executed (constructor included), the super-class is completely initialized already

Example

```
class ArtWork {
  ArtWork() {
    System.out.println("ctor ArtWork"); }
}
```

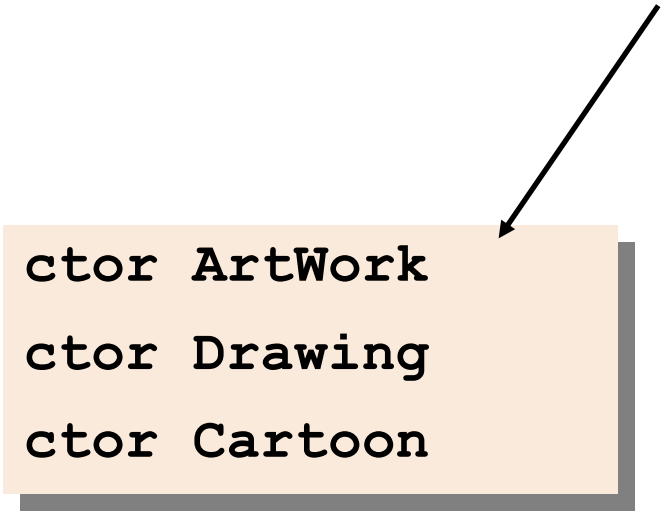
```
class Drawing extends ArtWork  {
  Drawing() {
    System.out.println("ctor Drawing"); }
}
```

```
class Cartoon extends Drawing {
  Cartoon() {
    System.out.println("ctor Cartoon"); }
}
```



Example

```
Cartoon obj = new Cartoon();
```



A word of advice

- Default constructor “disappears” if custom constructors are defined

```
class Parent{
  ???
  Parent(int i){}
}
class Child extends Parent{
}
// error!
```

```
class Parent{
 Parent(){} //explicit default
 Parent(int i){}
}
class Child extends Parent {
}
// ok!
```

A word of advice

- If constructors with arguments are defined in the base class …
- … and default constructor is not defined explicitly
- … the compiler cannot insert the call automatically
- In this case, child class constructor must call the right constructor of the parent class, explicitly

Example

```
class Employee {
  private String name;
  private double wage;
       ???
  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super();  // ERROR !!!
    unit = u;
  }
}
```

Example

```
class Employee {
  private String name;
  private double wage;

  Employee(String n, double w){
    name = n;
    wage = w;
  }
}
```

```
class Manager extends Employee {
  private int unit;

  Manager(String n, double w, int u) {
    super(n,w); // ok
    unit = u;
  }
}
```

Dynamic binding/ Polymorphism

SOftEng
http://softeng.polito.it

Polymorphism

- A reference of type T can point to an object of type S if-and-only-if
  - S is equal to T or
  - S is a subclass of T

```
Car myCar;
myCar = new Car();
myCar = new ElectricCar();
```

Example

```
Car[] garage = new Car[4];
garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] = new ElectricCar();
garage[3] = new Car();


for(int i=0; i<garage.length; i++){
   garage[i].turnOn();
}
```

Example

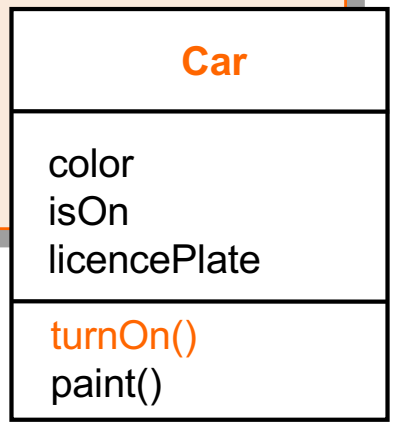
```
Car[] garage = new Car[4];
garage[0] = new Car();
garage[1] = new ElectricCar();
garage[2] = new ElectricCar();
garage[3] = new Car();

for(Car a : garage){
    a.turnOn();
}
```

Type checking

- The compiler performs a check on method invocation on the basis of the reference type
  - Does the type provide such a method?

```
for(Car a : garage){
    a.turnOn();
}
```

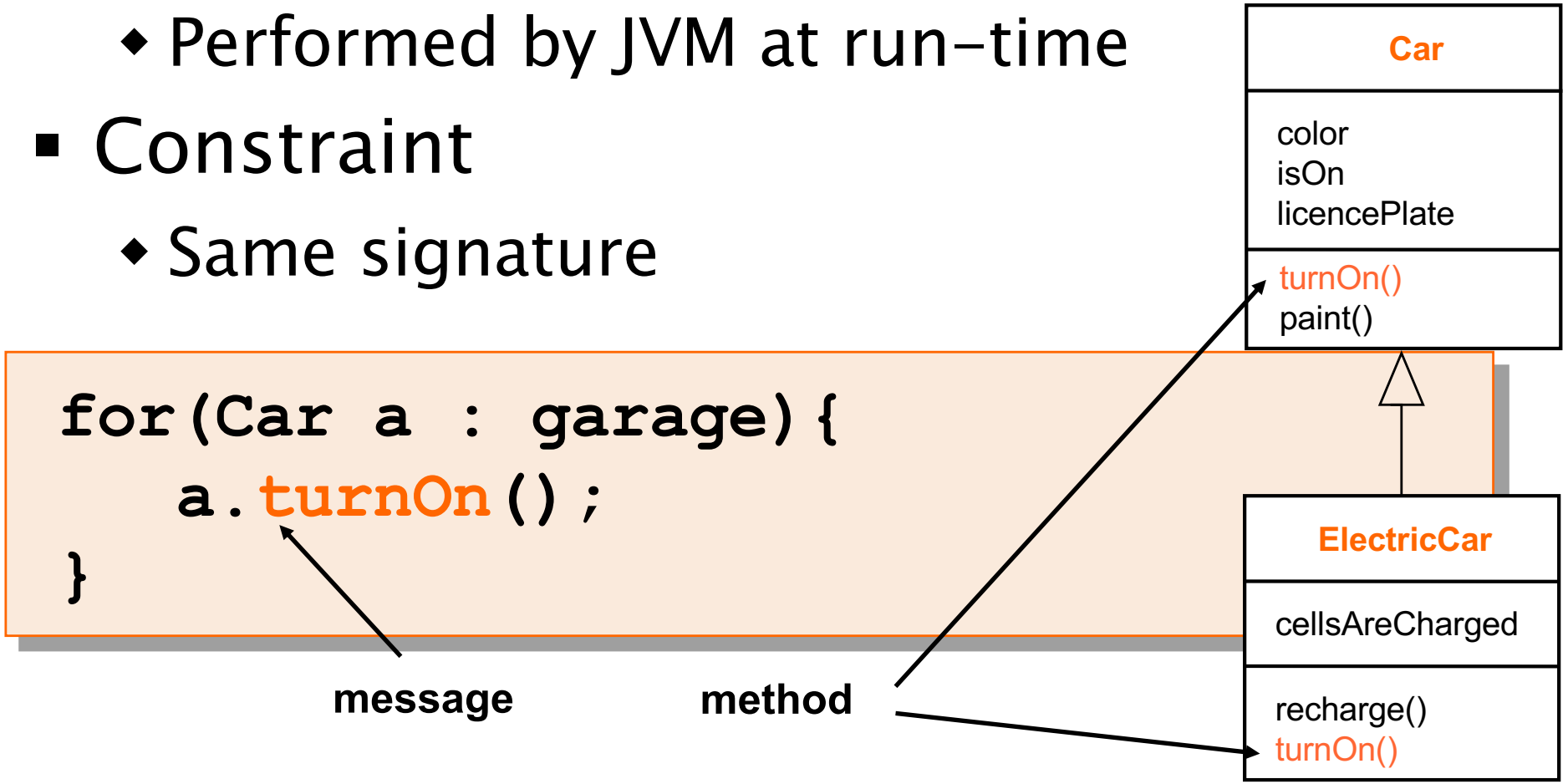
Does the type of a (i.e., `Car`) provide method `turnOn()`?



SOftEng
http://softeng.polito.it

Dynamic binding

- Association message/method
  - Performed by JVM at run-time
- Constraint
  - Same signature



Dynamic binding procedure

1. The JVM retrieves the effective class of the target object
2. If that class contains the invoked method it is executed
3. Otherwise the parent class is considered and second step is repeated

Note: the procedure is guaranteed to terminate

Why dynamic binding

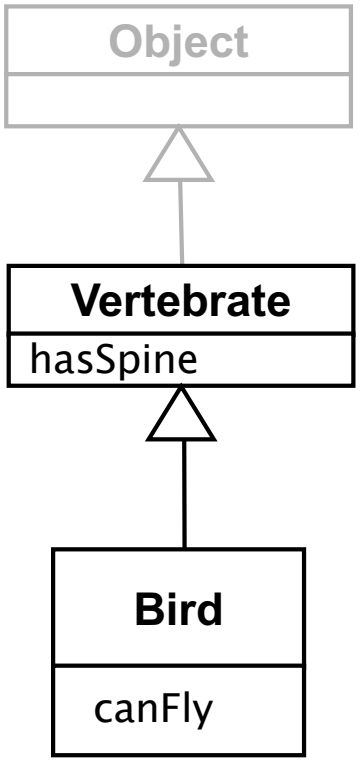
- Several objects from different classes, sharing a common ancestor class
  - Can be treated uniformly
- Algorithms can be written for the base class (using the relative methods) and applied to any subclass

Object

SOftEng
http://softeng.polito.it

Class Object

- **`java.lang.Object`**
- All classes are subtypes of **`Object`**



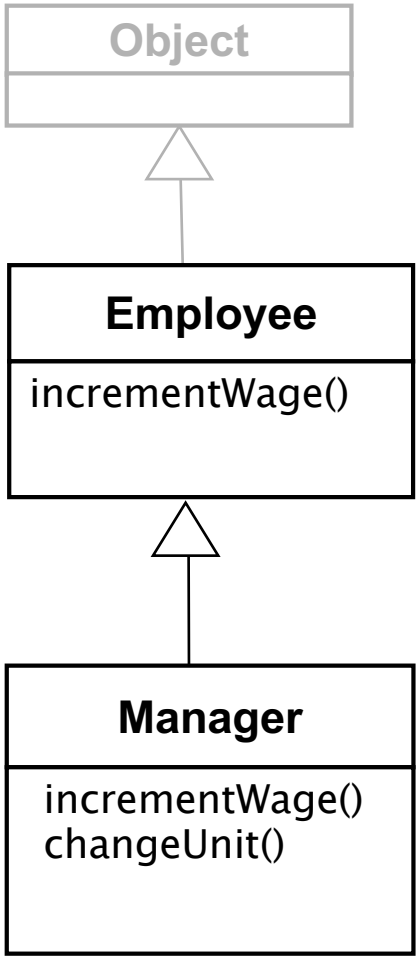
`extends Object` Implicitly

```
class Vertebrate  {
   …
}
```

```
class Bird extends Vertebrate{
   …
}
```

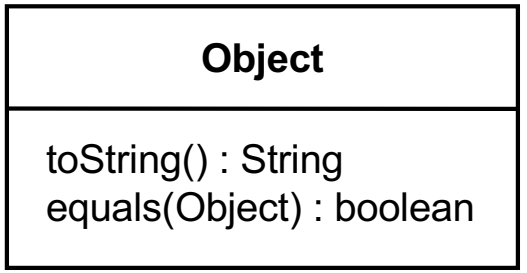
SoftEng
http://softeng.polito.it

Company


Object
Employee
incrementWage()
Manager
incrementWage()
changeUnit()

Why class Object

- Any instance of any class can be seen as an instance of class **`Object`**
- **`Object`**
  - Is the universal reference type
  - Object defines some common operations, which are useful for (and are inherited by) all classes
  - Often, they are overridden in sub-classes



Groups of objects

- References of type **`Object`** play a role similar to **`void*`** in C

```
Object [] objects = new Object[3];
objects[0] = "First!";
objects[2] = new Employee("Luca","Verdi");
objects[1] = new Integer(2);
for(Object obj : objects){
  System.out.println(obj);
}
```

Wrappers must be used instead of primitive types

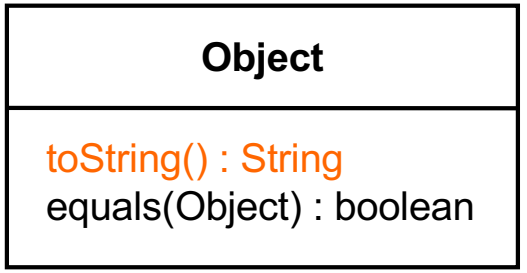
Methods of class Object

- **`toString()`**
  - Returns a string uniquely identifying the object
  - Default implementation returns:

`ClassName@######`

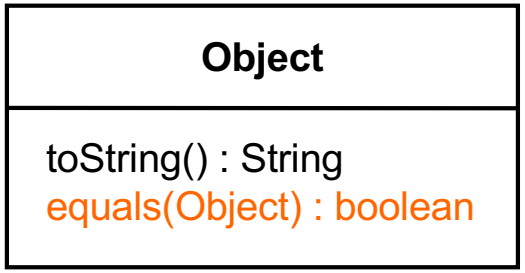
  - Es:

`org.Employee@af9e22`



Methods of class Object

- **`equals()`**
  - Tests equality of values
  - Default implementation compares references:




```
public boolean equals(Object other){
return this == other;
}
```

  - Must be overridden to compare contents, e.g.:

```
public boolean equals(Object o){
Student other = (Student)o;
return this.id.equals(other.id);
}
```

System.out.print(Object)

- Print methods implicitly invoke toString() on all object parameters

```
class Car{ String toString(){…} }
Car c = new Car();
System.out.print(c);  // same as...
System.out.print(c.toString());
Object ob = c;

System.out.print(ob); // Car's toString()
                      // called
```

Casting

SOftEng
http://softeng.polito.it

Types

- Java is a strictly typed language, i.e., each variable has a type

```
float f;
f = 4.7;   // legal
f = "string"; // illegal

Car c;
c = new Car();  // legal
c = new String(); // illegal
```

Cast

- Type conversion
  - Explicit or implicit

```
int i = 44;
float f = i;
// implicit cast 2'c -> fp
f = (float) 44;
// explicit cast
```

Cast – Gener/Specialization

- Things slightly change with inheritance
- Normal case

```
class Car{};
class ElectricCar extends Car{};
Car c = new Car();
ElectricCar ec = new ElectricCar ();
```

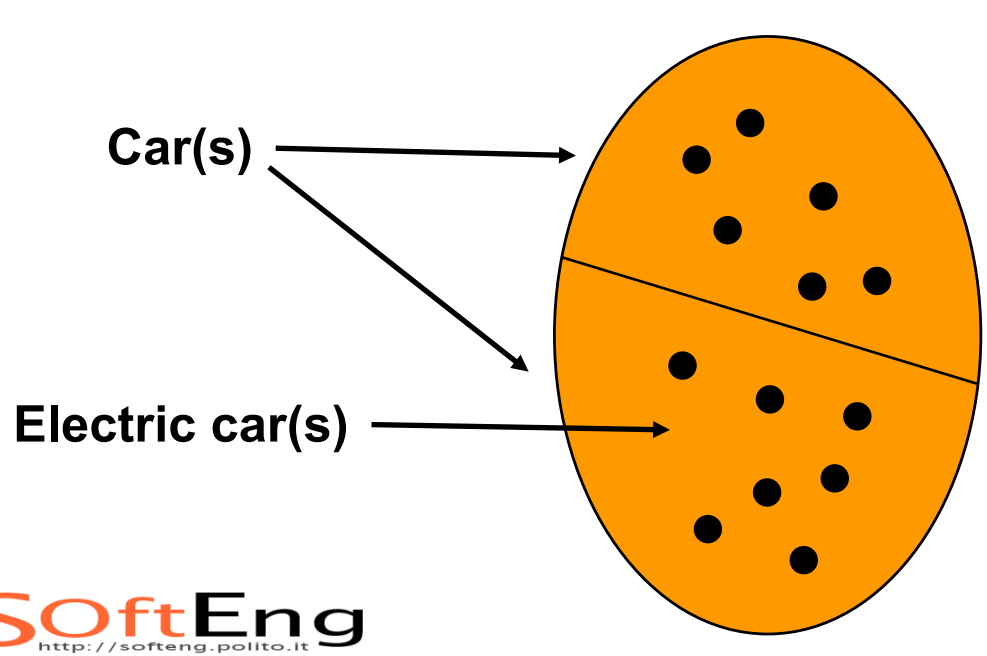
Cast – Gener/Specialization

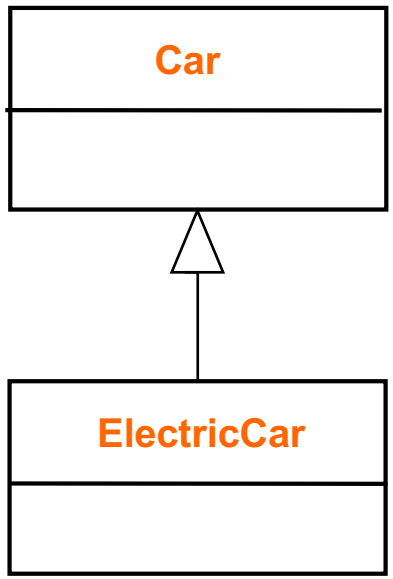
- New case

```
class Car{};
class ElectricCar extends Car{};
Car a = new ElectricCar (); // legal??
```

Cast – Gener/Specialization

- Legal!
  - Specialization defines a sub-typing relationship
  - **`ElectricCar`** type is a subset of **`Car`** type





Upcast

- Assignment from a more specific type (subtype) to a more general type (supertype)

```
class Car{};
class ElectricCar extends Car{};
Car c = new ElectricCar();
```

- Note well: reference type and object type are separate concepts
  - Object referenced by `c` continues to be of `ElectricCar` type

Upcast

- It is automatic
  - It is always true that an **`ElectricCar`** is a **`Car`** too
  - Upcasts are always type-safe and are performed implicity by the compiler (though it is ok to add them explicity)

```
Car c = new Car();
ElectricCar ec = new ElectricCar();
c = ec;
```

Object type does not change

SOftEng
http://softeng.polito.it

Downcast

- Assignment from a more general type (super-type) to a more specific type (sub-type)
  - As for upcast, reference type and object type do not change
- But, downcast must be explicit
  - It is a risky operation, no automatic conversion provided by the compiler (it is up to the programmer: it is the programmer that ensures that types are compatible, otherwise error)

Downcast – Example

```
Car c = new ElectricCar(); // implic. upcast

c.recharge(); // wrong!

// explicit downcast
ElectricCar ec = (ElectricCar)c;

ec.recharge(); // ok
```

Programmer knows they are compatible types (compiler trusts the programmer)

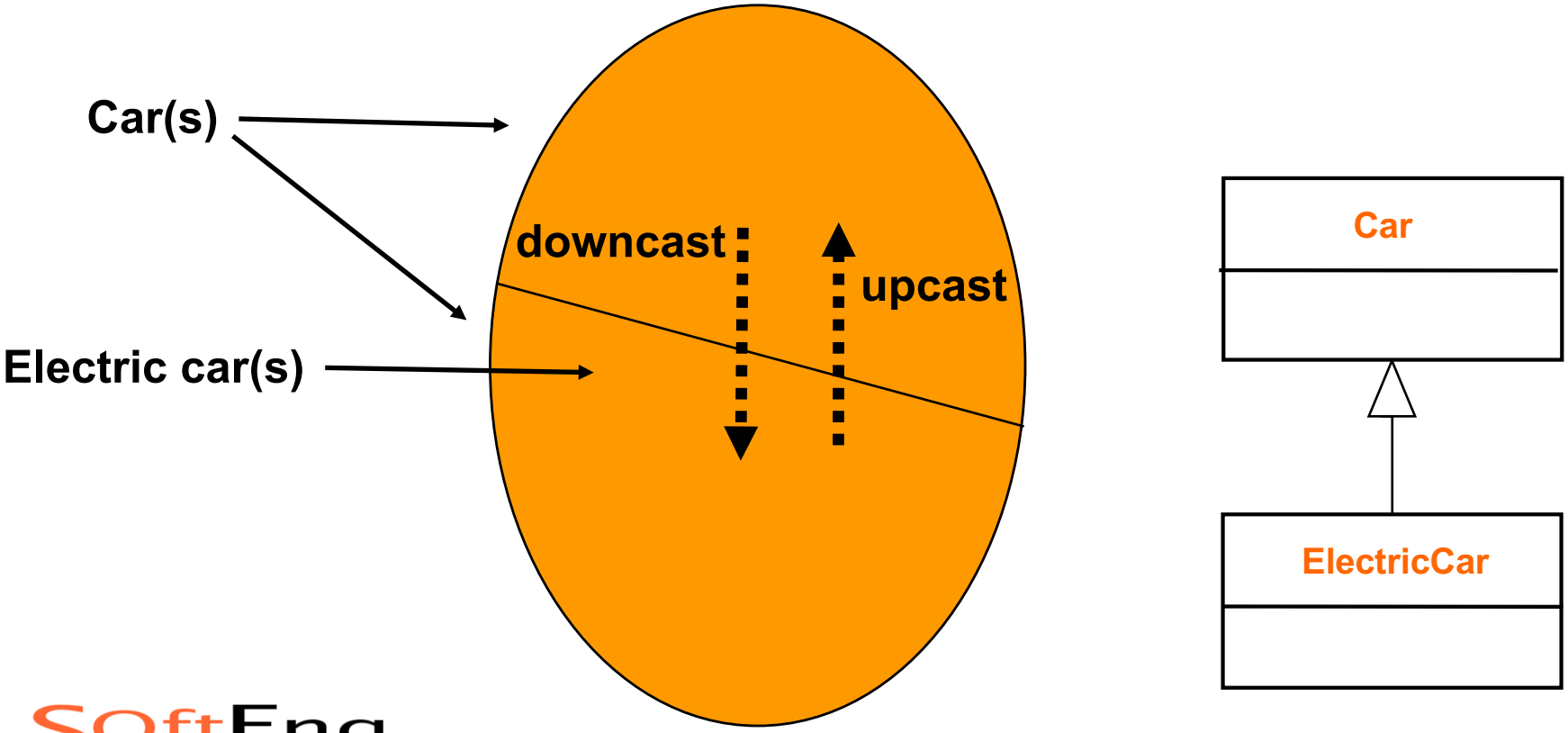
Downcast – Example

```
Car c = new Car();

c.recharge(); // wrong!

// explicit downcast
ElectricCar ec = (ElectricCar)c; //run-time
                                 //error


ec.recharge(); // wrong!
```

Programmer may be wrong (risk)

Visually

- All electric cars are cars too …
- But not all cars are electric cars are too



Downcast safety

- Use the `instanceof` operator
  - Returns true if the object referred to by the reference can be cast to the class
    - (I.e., if the object belongs to the given class or to any of its subclasses)

```
Car c = new Car();
ElectricCar ec;
if (c instanceof ElectricCar ){
   ec = (ElectricCar) c;
   ec.recharge();
}
```

Upcast to Object

- Each class is either directly or indirectly a subclass of **`Object`**
- It is always possible to upcast any instance to **`Object`** type

```
AnyClass foo = new AnyClass();
Object obj;
obj = foo;
```

Abstract classes

SOftEng
http://softeng.polito.it

Abstract class

- Often, a superclass is used to define common behavior for many children classes
  - Though some methods have no obvious implementation in the superclass (the class is too general to be instantiated)
  - The behavior is left partially unspecified
  - The superclass cannot be instantiated

Abstract class

- The `abstract` modifier marks the method as non-complete / undefined
- The modifier must be applied to all incomplete method and to the class

Abstract modifier

```
public abstract class Shape {

   private int color;

   public void setColor(int color){
      this.color = color;
   }

   // to be implemented in child classes
   public abstract void draw();
}
```

No method body

Abstract modifier

```
public class Circle extends Shape {

    public void draw() {
        // body goes here
    }
}


Shape a = new Shape(); // illegal: abstract
Shape a = new Circle();// ok
```

Interfaces

SOftEng
http://softeng.polito.it

Java interface

- A special type of class where
  - Methods are implicitly abstract (no body)
  - Attributes are implicitly static and final
  - Members are implicitly public
- Defined with keyword `interface`
- Cannot be instantiated (no **`new`**)
- Can be used to define references
  - Similar to abstract class

Interface implementation

- Class **implements** interfaces
  - A class must implement all interface methods unless the class is abstract
  - Interfaces are similar to abstract classes with only abstract methods

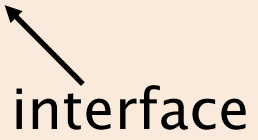
```
interface Iface {
 void method();
}
```

```
class Cls implements Iface
{
 void method(){
  // ...
 }
}
```

Rules (interface)

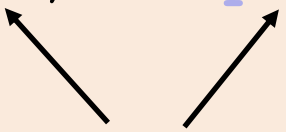
- An interface can extend another interface, cannot extend a class

```
interface Bar extends Comparable {
  void print();
}
```



- An interface can extend multiple interfaces

```
interface Bar extends Orderable, Comparable{
  ...
}
```

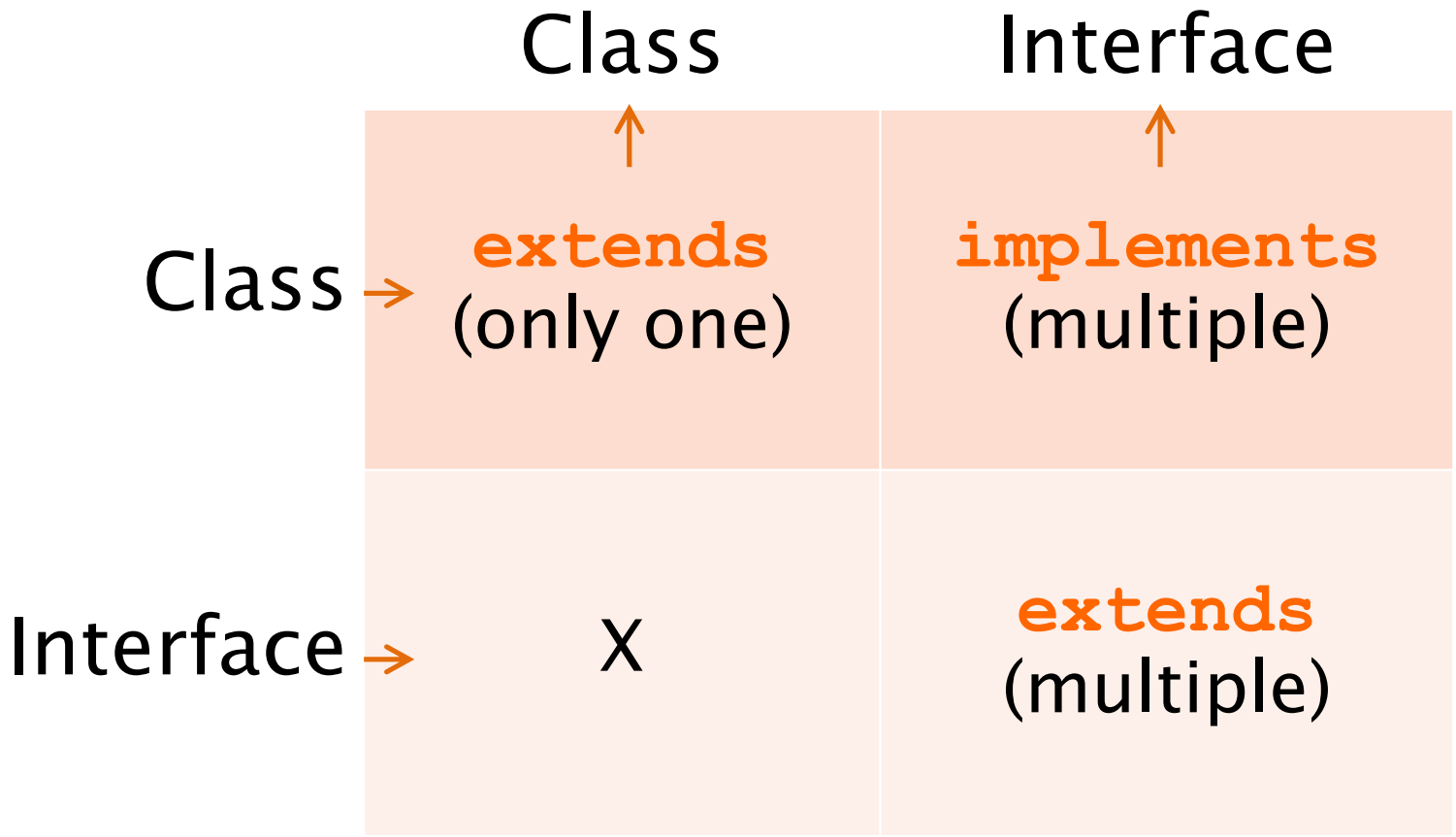


Rules (class)

- A class can extend only one class
- A class can implement multiple interfaces

```
class Employee
   extends Persona
   implements Orderable, Comparable {…}
```

Rules


Class
Interface
Class
extends
(only one)
implements
(multiple)
Interface
X
extends
(multiple)

Purpose of interfaces

- Define a common “interface”
  - That can be implemented in different ways
- Provide common behavior
  - By defining a (set of) method(s) that can be called by algorithms
- Enable communication decoupling
  - Define a set of callback method(s)

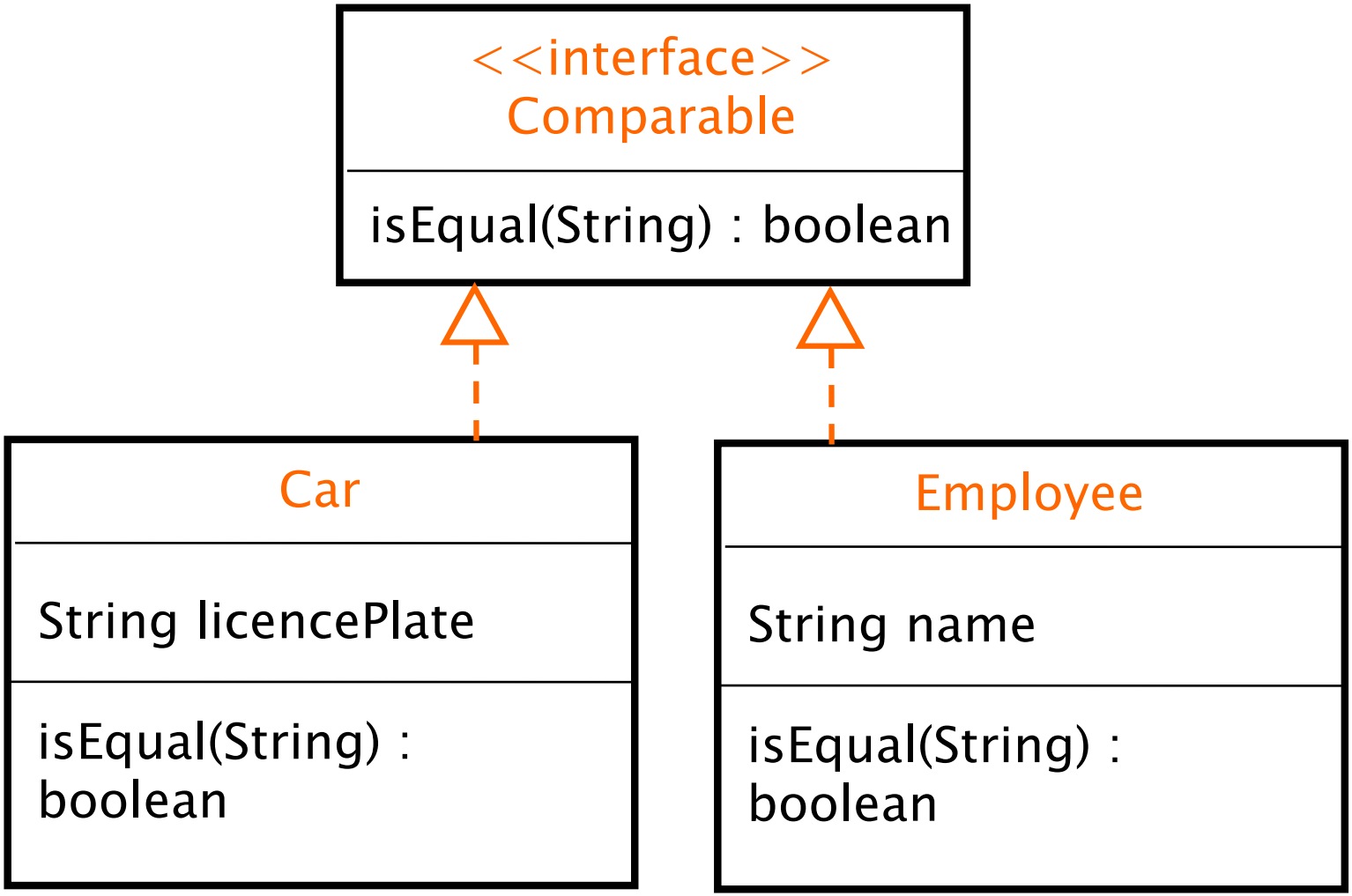
Alternative implementations

- Complex numbers

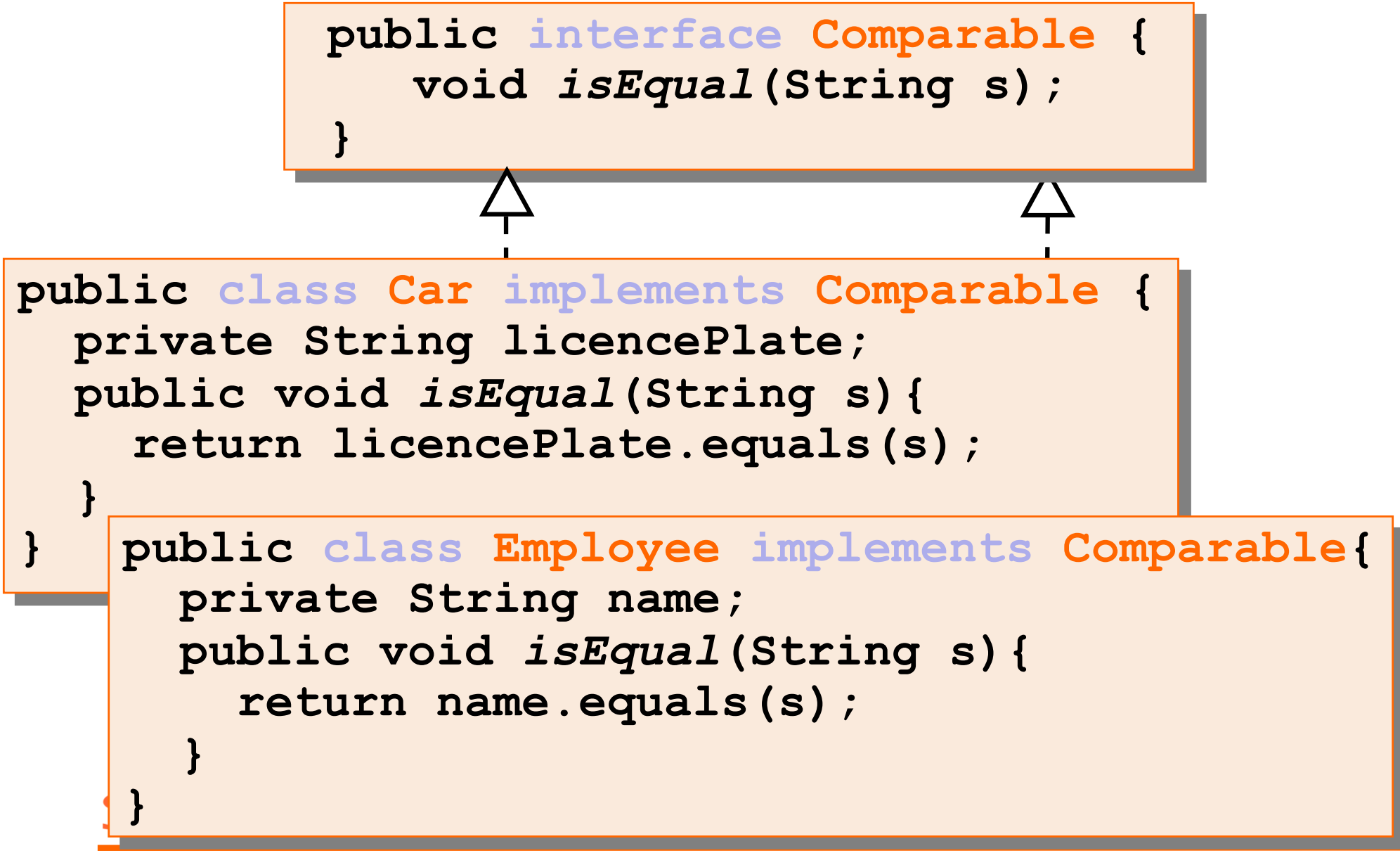
```
public interface Complex {
   double real();
   double imaginary();
   double modulus();
   double argument();
}
```

- Can be implemented using either Cartesian or polar coordinates

Example


<<interface>>
Comparable
isEqual(String) : boolean
Car
String licencePlate
isEqual(String) : boolean
Employee
String name
isEqual(String) : boolean

Example


public interface Comparable {
void isEqual(String s);
}
public class Car implements Comparable {
private String licencePlate;
public void isEqual(String s){
return licencePlate.equals(s);
}
}
public class Employee implements Comparable{
private String name;
public void isEqual(String s){
return name.equals(s);
}
}

A word of advice

- Defining a class that contains abstract methods only …
  - Is not illegal
  - But one should use interfaces instead
- Overriding methods in subclasses can maintain or extend the visibility of overridden superclass's methods
  - E.g. protected `int m()` can't be overridden by
    - `private int m()`
    - `int m()`
  - Only protected or public are allowed

Wrap-up session

- Inheritance
  - Objects defined as sub-types of already existing objects; they share the parent data/methods without having to re-implement
- Specialization
  - Child class augments parent (e.g. adds an attribute/method)
- Overriding
  - Child class redefines parent method
- Implementation/reification
  - Child class provides the actual behaviour of a parent method

Wrap-up session

- Polymorphism
  - The same message can produce different behavior depending on the actual type of the receiver objects (late binding of message/method)