

Inheritance



Version 3 - May 2009

Overriding

```
▪ class Vector{
    int vect[];
    void add(int x) {...}
}
▪ class OrderedVector extends Vector{
    void add(int x){...}
}
```

SoftEng

Inheritance

- A class can be a sub-type of another class
- The inheriting class contains all the methods and fields of the class it inherited from plus any methods and fields it defines
- The inheriting class can **override** the definition of existing methods by providing its own implementation
- The code of the inheriting class consists only of the changes and additions to the base class

SoftEng

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
        System.out.println(name); //un-optimized!
        System.out.println(managedUnit);
    }
}
Employee e1 = new Employee();
Employee e2 = new Manager();
e1.print(); // name
e2.print(); // name and unit
```

SoftEng

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
```

SoftEng

Inheritance and polymorphism

- Employee e1 = new Employee();
- Employee e2 = new Manager(); //ok, is_a
- e1.print(); // name
- e2.print(); // name and unit

SoftEng

Why inheritance

- Frequently, a class is merely a modification of another class. In this way, there is minimal 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
 - ♦ Less chances of different (and inconsistent) implementations of the same operation

SoftEng

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

SoftEng

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 PhoneBook
 - ♦ Add mail address and cell number

SoftEng

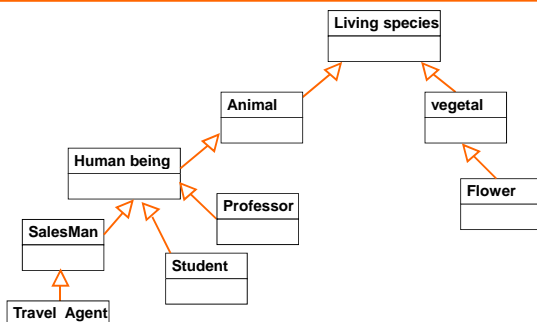
Inheritance in few words

- Subclass
 - ♦ Inherits attributes and methods
 - ♦ Can modify inherited attributes and methods (override)
 - ♦ Can add new attributes and methods

SoftEng

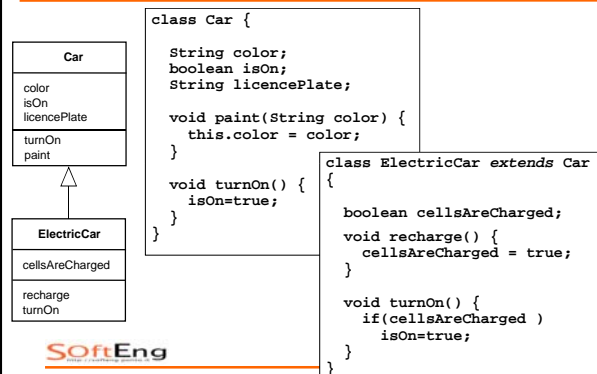
11

Example of inheritance tree



SoftEng

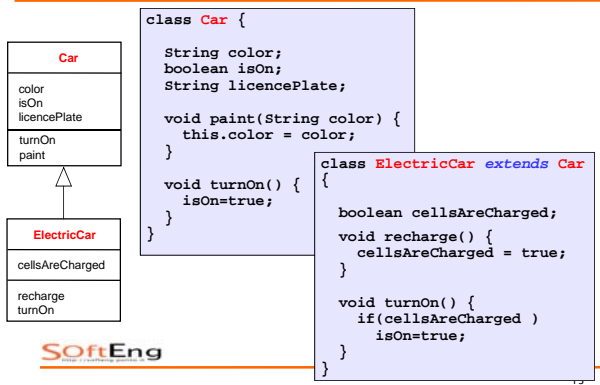
Inheritance in Java: extends



SoftEng

12

Inheritance in Java: *extends*



Example

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

class Manager extends Employee {

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

Arrows point from the text "Not visible" to the `name` and `wage` variables in the `print()` method of the `Manager` class, indicating that these private attributes are not visible to the subclass.

ElectricCar

- Inherits
 - ♦ attributes (`color`, `isOn`, `licencePlate`)
 - ♦ methods (`paint`)
- Modifies (overrides)
 - ♦ `turnOn()`
- Adds
 - ♦ attributes (`cellsAreCharged`)
 - ♦ Methods (`recharge`)

Protected

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

Visibility (scope)



In summary

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

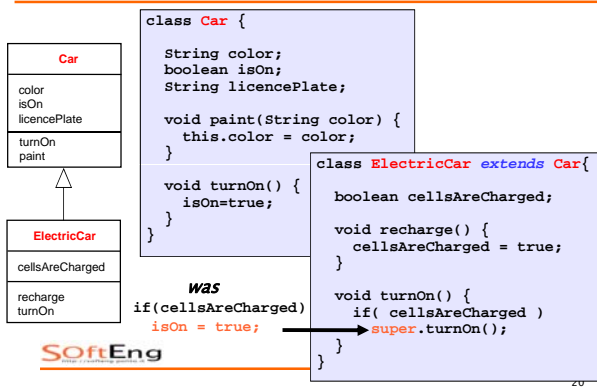
Super (reference)

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

Inheritance and constructors



Example



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
- The call is inserted as the **first** statement of each child constructor

Attributes redefinition

- ```
Class Parent{
 protected int attr = 7;
}
```
- ```
Class Child{
    protected String attr = "hello";

    void print(){
        System.out.println(super.attr);
        System.out.println(attr);
    }

    public static void main(String args[]){
        Child c = new Child();
        c.print();
    }
}
```

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("New ArtWork");
    }
}

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

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



25

Super

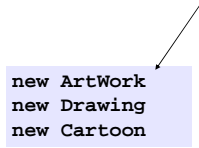
- If you define custom **constructors with arguments**
 - and default constructor is not defined explicitly
- the compiler cannot insert the call automatically

SoftEng

28

Example (cont'd)

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



```
new ArtWork
new Drawing
new Cartoon
```

SoftEng

26

Super

- Child class constructor must call the right constructor of the parent class, **explicitly**
- Use **super()** to identify constructors of parent class
- **First** statement in child constructors

SoftEng

29

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

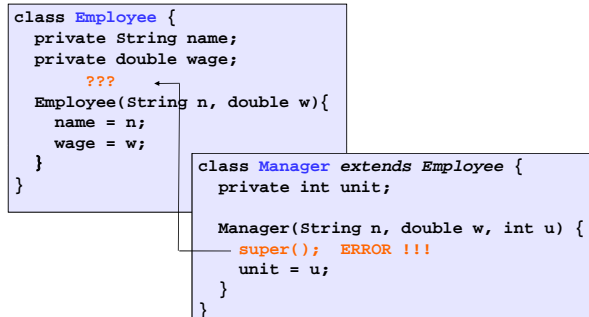
SoftEng

27

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;
    }
}
```



SoftEng

30

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);
        unit = u;
    }
}
```

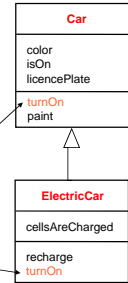
Binding

- Association message/method
- Constraint
 - ♦ Same signature

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

message

method



Dynamic binding/ polymorphism

Object

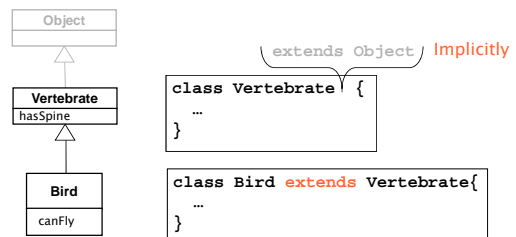
Example

- `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();
}
```

## Java Object

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



## Java Object

- Each instance can be seen as an **Object instance** (see Collection)
- Object defines some **services**, which are useful for all classes
- Often, they are **overridden** in sub-classes

| Object                                          |
|-------------------------------------------------|
| toString() : String<br>equals(Object) : boolean |

## Java 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);
}
```

| Object                                          |
|-------------------------------------------------|
| toString() : String<br>equals(Object) : boolean |

## Objects' collections

- 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

## 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());
```
- Polymorphism applies when **toString()** is overridden

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

## Java Object

- **toString()**
  - ♦ Returns a string uniquely identifying the object
  - ♦ Default implementation returns:  
**ClassName@#####**
  - ♦ Es:  
`org.Employee@af9e22`

| Object                                          |
|-------------------------------------------------|
| toString() : String<br>equals(Object) : boolean |

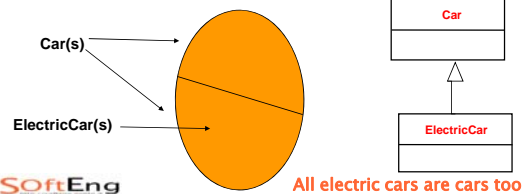
## Casting

## 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`

## Specialization – 3

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



## Specialization

- Things change slightly
- Normal case...

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

## Cast

- Type conversion (explicit or implicit)

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

## Specialization – 2

- New case...

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

## Upcast

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

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

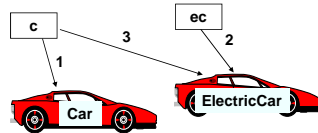
- **Note well** – reference type and object type are separate concepts
  - ♦ Object referenced by 'c' continues to be of ElectricCar type

## Upcast

- It is dependable
  - It is always true that an electric car is a car too
- It is automatic

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

Up-casting:  
Object type does NOT change



## Dowcast - Example II

```
Car c = new Car();
c.recharge(); // wrong!
// explicit dowcast
ElectricCar ec = (ElectricCar)c;
ec.recharge(); // wrong!
```

Run-time error

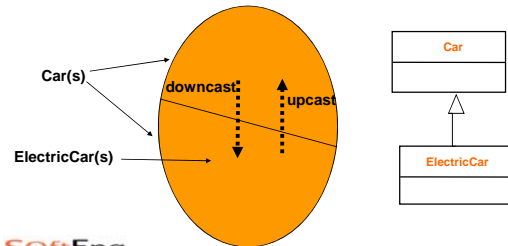
**YOU might be wrong (risk)**

## Dowcast

- Assignment from a more general type (super-type) to a more specific type (sub-type)
  - As above, reference type and object type do not change
- MUST** be explicit
  - It's a risky operation, no automatic conversion provided by the compiler (it's up to you!)

## Visually

- All electric cars are cars too
- Not all cars are electric cars too



## Dowcast - Example I

```
Car c = new ElectricCar(); // impl. upcast
c.recharge(); // wrong!
// explicit dowcast
ElectricCar ec = (ElectricCar)c;
ec.recharge(); // ok
```

**YOU know they are compatible types (compiler trusts you)**

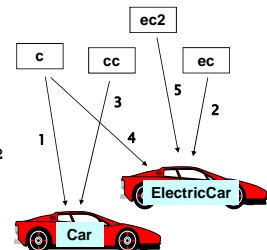
## Messy example

```
Car c, cc;
ElectricCar ec, ec2;
c = new Car (); // 1
c.recharge(); // NO

ec = new ElectricCar (); // 2
ec.recharge() // ok

cc = c; // 3

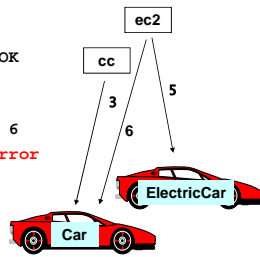
c = ec; // 4 Upcasting
c.recharge(); // NO
```



## Messy example (cont'd)

```
ec2 = c; // NO Downcast
ec2 = (ElectricCar) c; // 5, OK
ec2.recharge(); // OK

ec2 = (ElectricCar) cc; // 6
ec2.recharge(); // runtime error
```



## Abstract classes



## Avoid wrong down-casting

- Use the **instanceof** operator
- `Car c = new Car();`  
`ElectricCar ec;`
- ```
if (c instanceof ElectricCar) {
    ec = (ElectricCar) c;
    ec.recharge();
}
```

was
`((ElectricCar)c).recharge();`

Abstract class

- Often, superclass is used to define common behavior for many child classes
- But the class is too general to be instantiated
- Behavior is partially left unspecified (this is more concrete than interface)

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 (see `Collection`)

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

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  
    }  
}
```

```
Object a = new Shape(); // Illegal: abstract  
Object a = new Circle(); // OK
```

Purpose of interfaces

- Define an “interface” that can be implemented in different ways
- Provide a (set of) method(s) that can be called by algorithms
- Define a (set of) callback method(s)

Interface

Alternative implementations

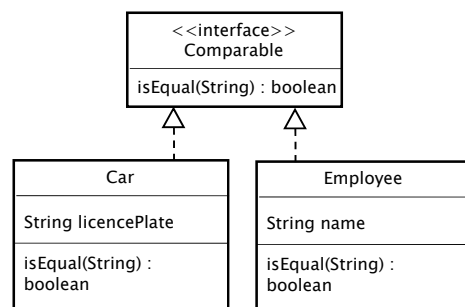
- Complex numbers

```
public interface Complex {  
    double real();  
    double imaginary();  
    double modulus();  
    double argument();  
}
```
- Can be implemented using either Cartesian or polar coordinates

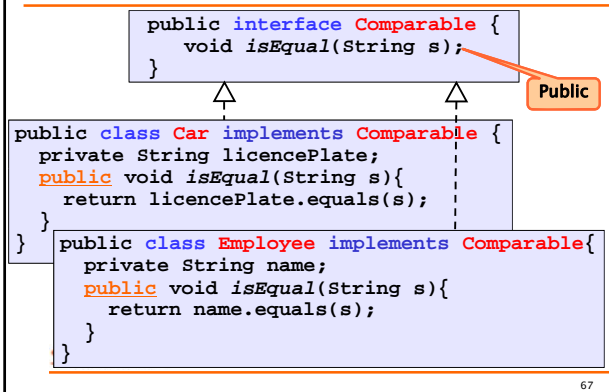
Java interface

- An interface is a special type of “class” where **methods and attributes** are implicitly **public**
 - ♦ **Attributes** are implicitly **static and final**
 - ♦ **Methods** are implicitly **abstract** (no body)
- **Cannot** be instantiated (no new)
- **Can** be used to define references

Example



Example (cont'd)



Rules (class)

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

```
class Person
    extends Employee
    implements Orderable, Comparable {...}
```

SoftEng

70

Example

```
public class Foo {
    private Comparable objects[];
    public Foo(){
        objects = new Comparable[3];
        objects[0] = new Employee();
        objects[1] = new Car();
        objects[2] = new Employee();
    }
    public Comparable find(String s){
        for(int i=0; i< objects.length; i++){
            if(objects[i].isEqual(s))
                return objects[i];
        }
    }
}
```

SoftEng

68

A word of advice

- Defining a class that contains abstract methods only is not illegal but..
 - ♦ You 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

SoftEng

71

Rules (interface)

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

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

interface

- An interface **can extend multiple interfaces**

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

interfaces

SoftEng

69

Homework

- See the doc of java.lang.Comparable

```
public interface Comparable{
    int compareTo(Object obj);
}
```

- Returns a negative integer, 0, or a positive integer as this object is less than, equal, or greater than obj

SoftEng

72

Homework (cont'd)

- Define **Employee**, which implements Comparable (order by ID)
- Define **OrderedArray** class
 - ♦ void add(Comparable c) //ordered insert
 - ♦ void print() //prints out
- Test it with the following main

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)

Homework (cont'd)

```
public static void main(String args[]){  
    int size = 3; // array size  
    OrderedArray oa = new OrderedArray(size);  
  
    oa.add( new Employee("Mark", 37645) );  
    oa.add( new Employee("Andrew", 12345) );  
    oa.add( new Employee("Sara", 97563) );  
  
    oa.print();  
}
```

Name

ID

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)

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