Inheritance

Version 2 - June 2008

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

Example

- 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

Overriding

- Class Vector{
  int vect[20];
  void add(int x) {…}
  }
- Class OrderedVector extends Vector{
  void add(int x){…}
  }

SOftEng

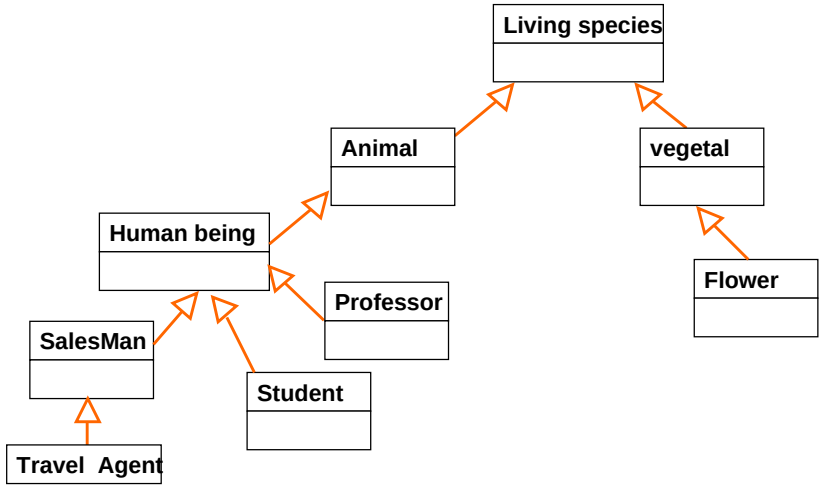
Why inheritance

- 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

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

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

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

Inheritance and polymorphism

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

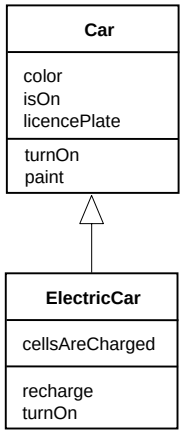
SOftEng

Inheritance in few words

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

SOftEng

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
{

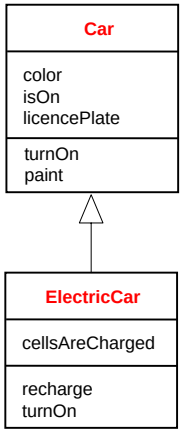
  boolean cellsAreCharged;

  void recharge() {
    cellsAreCharged = true;
  }

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

SOftEng

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)
  - turnOn()
- Adds
  - attributes (cellsAreCharged)
  - Methods (recharge)

Visibility (scope)

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 class only
  - **protected** are accessible within the class and its subclasses

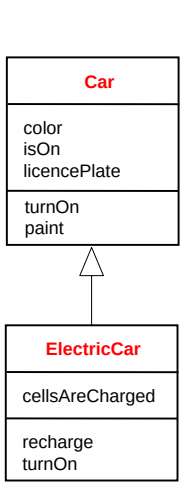
In summary

	Method in the same **class**	Method of another class in the same **package**	Method of **subclass**	Method of 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

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();
  }
}
```

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();
   }
}
```


Inheritance and constructors

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

Construction of child objects

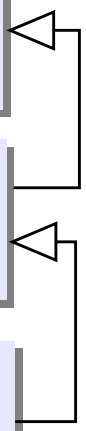
- Execution of constructors proceeds top-down in the inheritance hierarchy
- In this way, when a method of the child class is executed (constructor included), the super-class is completely initialized already

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"); }
}
```



Example (cont'd)

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

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


Super

- If you define custom constructors with arguments
- and default constructor is not defined explicitly

➔ the compiler cannot insert the call automatically

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

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

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;
  }
}
```

SOftEng

Dynamic binding/ polymorphism

SOftEng
http://softeng.polito.it

Example

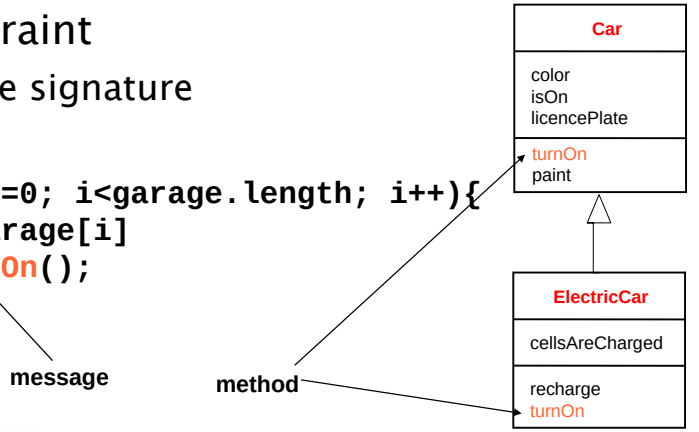
- `Car[] garage = new Car[4];`
- `garage[0] = new Car();`
- `garage[1] = new ElectricCar();`
- `garage[2] = new ElectricCar();`
- `garage[3] = new Car();`

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


Binding

- Association message/method
- Constraint
  - Same signature

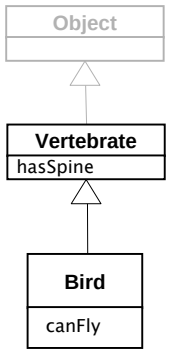
```
Car a;
for(int i=0; i<garage.length; i++){
   a = garage[i]
   a.turnOn();
}
```



Object

Java Object

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



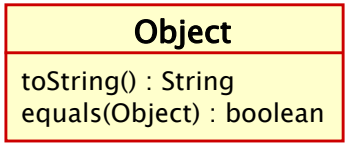
`extends Object` Implicitly

```
class Vertebrate {
  …
}
```

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

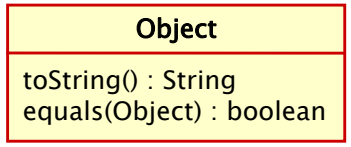
Java Object

- 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



Java Object

- **`toString()`**
  - Returns a string uniquely identifying the object
- **`equals()`**
  - Tests equality of values



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() is 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
  ```


Specialization

- Things change slightly
- Normal case…

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

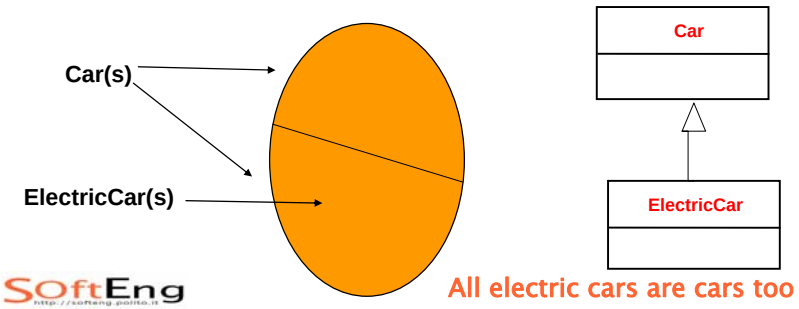

Specialization – 2

- New case…

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


Specialization – 3

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



All electric cars are cars too

Cast

- Type conversion (explicit or implicit)

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

Upcast

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

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

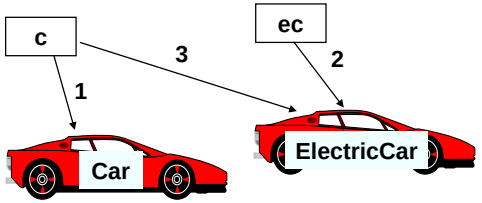
- Note well – reference type and object type are separate concepts
  - Object referenced by 'c' continues to be of ElectricCar type

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



Downcast

- 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!)

Downcast – Example I

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

c.recharge(); // wrong!

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

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

YOU know they are compatible types (compiler trusts you)

Dowcast – Example II

```
Car c = new Car();

c.rechage(); // wrong!

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

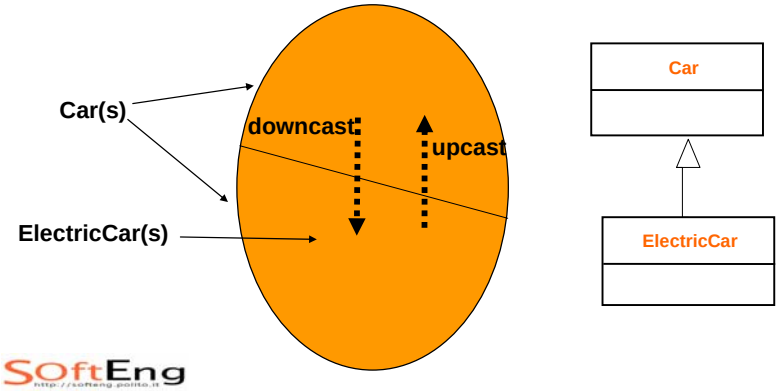
ec.recharge(); // wrong!
```

Run-time error

YOU might be wrong (risk)

Visually

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



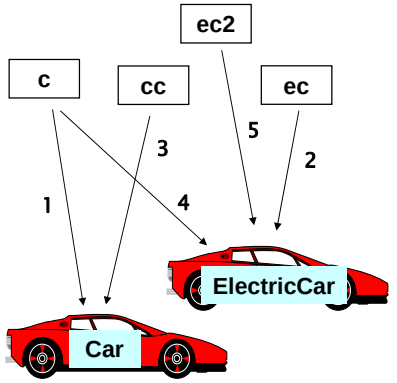
Messy example

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

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

cc = c; // 3

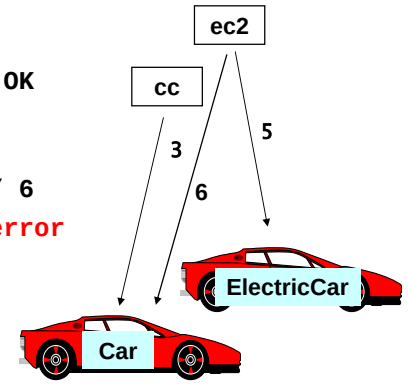
c= ec; // 4 Upcasting
c. recharge(); // NO
```



Messy example (cont'd)

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

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



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();`

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 classes

SOftEng
http://softeng.polito.it
```

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)

Abstract modifier

```
public abstract class Shape {

   privte 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
   }
}

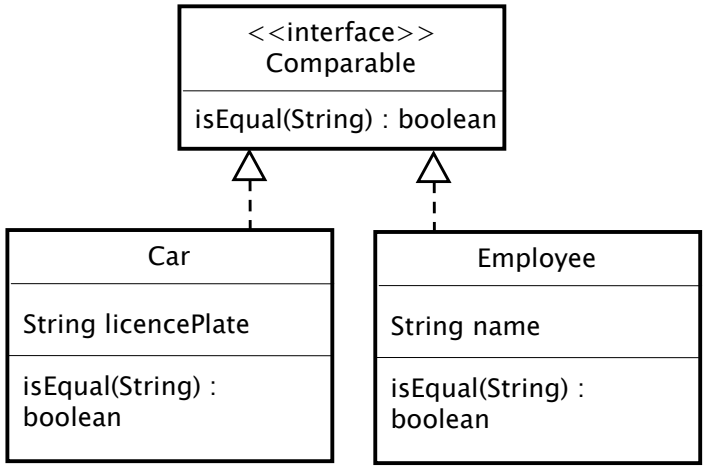

Object a = new Shape();  // Illegal
Object a = new Circle(); // OK
```


Interface

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)

```
public interface Comparable {
   void isEqual(String s);
}
```

Public

```
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);
  }
}
```


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];
  }
}
```

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

Rules (class)

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

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

SOftEng

A word of advice

- Defining a class that contains abstract methods only is not illegal
  - 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

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

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

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

- 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)

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)