# OOP Inheritance

Università di Modena e Reggio Emilia
*Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)*

# Motivations

- They all move, have a shape, shields and weapons. Can they share the same code?

# Motivation

- Frequently, a class is merely a modification of another class. Inheritance allows 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

- 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

# Inheritance (Real Life)

- A new design created by the modification of an already existing design
  - The new design consists of only the changes

# Example I

```
Class Car {
    boolean isOn;
    string licensePlate;

    void turnOn() {…}
    void turnOff() {…}
}
```

```
Class SDCar extends Car {
    boolean isCharged;
    boolean isSelfDriving;

    void turnSDOn() {…}
    void turnSDOff() {…}
}

SDCar c = new SDCar();
c.turnOn();      //OK!
c.tunrSDOn();   //OK!
```

# Example II (Override)

```
Class Car {
    boolean isOn;
    string licensePlate;

    void turnOn() {…}
    void turnOff() {…}
}
```

```
Class SDCar extends Car {
    boolean isCharged;
    boolean isSelfDriving;

    /* override */
    void turnOn() {
        turnSDOff();
        /* … */
    }

    void turnSDOn() {…}
    void turnSDOff() {…}
}
```

# The keyword extends

```
Class SDCar extends Car {
    boolean isCharged;
    boolean isSelfDriving;

    void turnSDOn() {…}
    void turnSDOff() {…}
}
```

# SDCar

- Inherits
  - attributes (isOn, licencePlate)
  - methods (turnOn, turnOff)
- Modifies (overrides)
  - turnOn
- Adds
  - attributes (isCharged, isSelfDriving)
  - Methods (turnSDOn, turnSDOff)

# 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)

# Visibility

```
Class Car {
        private boolean isOn;
        private string licensePlate;

        public void turnOn() {…}
        public void turnOff() {…}
}
```

```
Class SDCar extends Car {
        void print() {
                System.out.println(licencePlate);
                // Do not work!! Not visible!!
        }
}
```

# The keyword 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

```
Class Car {
        protected boolean isOn;
        protected string licensePlate;

        public void turnOn() {…}
        public void turnOff() {…}
}
```

```
Class SDCar extends Car {
        void print() {
                System.out.println(licencePlate);
                // OK!
        }
}
```

# 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* | ✓ | ✓ | ✓ | ✓ |

# The keyword super

```
Class Car {
    boolean isOn;
    string licensePlate;

    void turnOn() {…}
    void turnOff() {…}
}
```

```
Class SDCar extends Car {
    boolean isCharged;
    boolean isSelfDriving;

    /* override */
    void turnOn() {
        turnSDOff();
        super.turnOn();
    }

    void turnSDOn() {…}
    void turnSDOff() {…}
}
```

# The keyword super

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

# 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. If parent class disabled default constructor (by defining others with params) super must be called!

# 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

# Example

```
class Car {
    String carName;
    // Default constructor active!
}

Class SDCar extends Car {
    SDCar() {
        // OK!
    }
}
```

# Example

```
class Car {
    String carName;
    // Custom constructor. Disables default one
    Car(String carName) {
        this.carName = carName;
    }
}
Class SDCar extends Car {
    SDCar() {
        // ERROR here. No default constructor on car!
    }
}
```

# Example

```
class Car {
    String carName;
    // Custom constructor. Disables default one
    Car(String carName) {
        this.carName = carName;
    }
}
Class SDCar extends Car {
    SDCar() {
        super("Fiat"); // OK!
    }
}
```

# 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 Car{
   Car() { System.out.println("New Car"); }
}
Class SDCar extends Car{
   SDCar() { System.out.println("New SDCar"); }
}
class ECar extends SDCar {
   ECar() { System.out.println("New ECar"); }
}

ECar c = new ECar();  // Which output?
```

# Dynamic binding and polymorphism

```
Car[] garage = new Car[4];

garage[0] = new Car();
garage[1] = new SDCar();
garage[2] = new SDCar();
garage[3] = new Car();

for(Car c : garage) {
    c.turnOn();
    // which method is actually called?
}
```

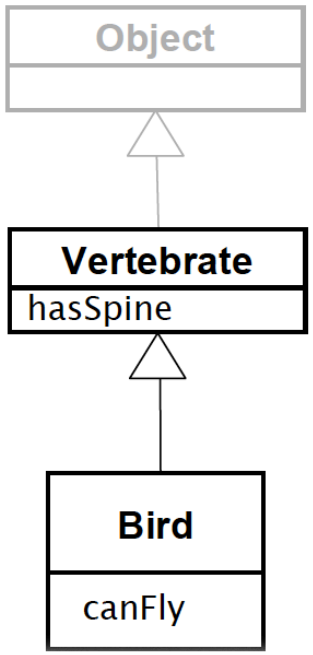
# Dynamic binding and polymorphism

- When using collections of objects belonging to a hierarchy of classes, methods actually called are known only at runtime.
- The same call (methods with the same signature) might have different results depending on the actual class of the object.

# Object

# Dynamic binding and polymorphism

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




```
class Vertebrate  {
   …
}
```

extends Object — Implicitly

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

# Java.lang.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

# toString(), equals()

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

# System.out.println(Object)

System.out.println() methods implicitly invoke toString() on all object parameters

```
class Car{
   String toString(){…}
}

Car c = new Car();
// equivalent calls
System.out.println(c);
System.out.println(c.toString());
```

Polymorphism applies when toString() is overridden

# 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

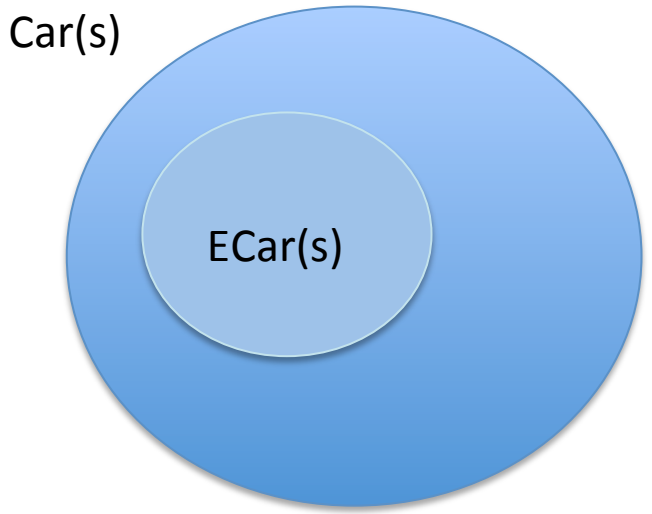
# Specialization

```
class Car {};
class ECar extends Car{};

Car c1 = new Car();     // OK!
Ecar c2 = new ECar (); // OK!

But also…

Car c3 = new Ecar();    // OK?
```

# Specialization

```
Car c3 = new Ecar();    // OK!
```

- Specialization defines a sub-typing relationship (is a ). ECar type is a subset of Car type.



# Upcasting

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

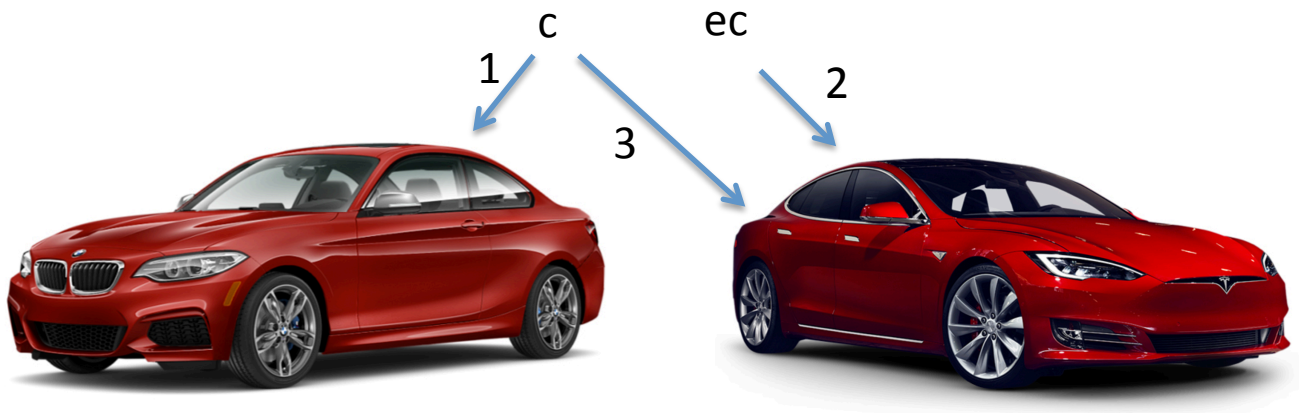
```
class Car{};
class ECar extends Car{};
Car c = new ECar ();
```

- Note well - reference type and object type are separate concepts. Object referenced by ‘c’ continues to be of ECar type! Only the interface changes!

# Upcasting

- It is dependable
  - It is always true that an electric car is a car too
- It is automatic (*e.g., float f = (int) 3*)

```
(1) Car c = new Car();
(2) ECar ec = new ECar ();
(3) c = ec;
```



# Example

```
Class Car {
        boolean isOn;
        string licensePlate;

        void turnOn() {…}
        void turnOff() {…}
}
```

```
Class ECar extends Car {
        boolean isCharged;
        boolean isSelfDriving;

        void turnSDOn() {…}
        void turnSDOff() {…}
}
```

```
ECar c1 = new ECar();
c1.turnSDOn() // OK!

Car c2 = c1;  // Upcast
c2.turnSDOn() // Invalid! (Car public interface does
not provide turnSDOn() call)
```

# Downcasting

- Assignment from a more general type (super-type) to a more specific type (sub-type)
  - 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!)

# Example

```
Class Car {
        boolean isOn;
        string licensePlate;

        void turnOn() {…}
        void turnOff() {…}
}
```

```
Class ECar extends Car {
        boolean isCharged;
        boolean isSelfDriving;

        void turnSDOn() {…}
        void turnSDOff() {…}
}
```

```
Car c1 = new ECar();
c1.turnSDOn() // Invalid!

ECar c2 = (Ecar)c1;  // Downcast
c2.turnSDOn() // OK!
```

# Example

```
Class Car {
        boolean isOn;
        string licensePlate;

        void turnOn() {…}
        void turnOff() {…}
}
```

```
Class ECar extends Car {
        boolean isCharged;
        boolean isSelfDriving;

        void turnSDOn() {…}
        void turnSDOff() {…}
}
```

```
Car c1 = new Car();
c1.turnSDOn() // Invalid!

ECar c2 = (Ecar)c1;  // Downcast
c2.turnSDOn() // Invalid! (Runtime!!)
```

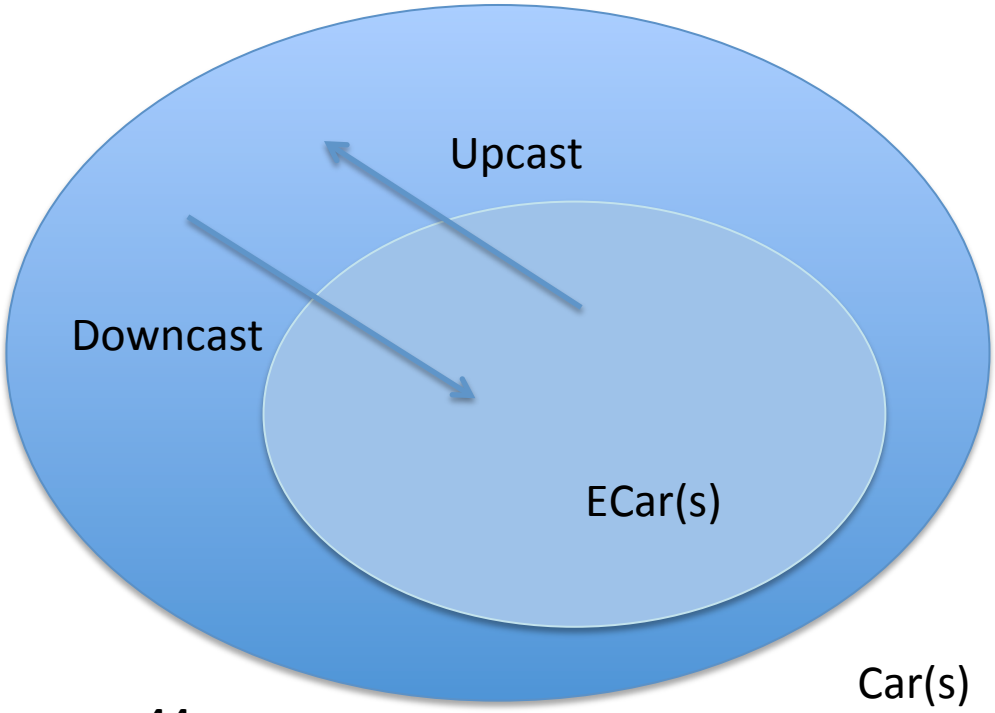
# Runtime is evil

- Compilers aid developers in writing working code. Runtime errors cannot be identified by compilers. Developers must be careful!
- Use the instanceof operator

```
Car c = new Car();
ECar ec;
if (c instanceof ECar){
   ec = (ECar) c;
   ec.turnSDOn();
}
```

# Specialization

- Specialization defines a sub-typing relationship (is a ). ECar type is a subset of Car type. All ECar(s) are Car(s). Not all Car(s) are ECar(s).



# 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 any = new AnyClass();
Object obj;
obj = (Object)any;
```

# Abstract Classes and Interfaces

# Abstract methods

- You can *declare* an object without *defining* it:

  Person p;

- Similarly, you can declare a *method* without defining it:

  public abstract void draw(int size);

  – Notice that the body of the method is missing

- A method that has been declared but not defined is an abstract method

# Abstract classes

- Any class containing an abstract method is an abstract class
- You must declare the class with the keyword abstract:

```
abstract class MyClass {...}
```

- An abstract class is *incomplete* (It has missing method bodies)
- You cannot instantiate (create a new instance of) an abstract class

# Abstract classes

- You can extend (subclass) an abstract class
  - If the subclass defines all the inherited abstract methods, it is concrete and can be instantiated
  - If the subclass does *not* define all the inherited abstract methods, it must be abstract too
- You can declare a class to be abstract even if it does not contain any abstract methods
  - This prevents the class from being instantiated

# Why have abstract classes?

- Suppose you wanted to create a class Shape, with subclasses Oval, Rectangle, Triangle, Hexagon, etc. You don't want to allow creation of a "Shape"
  - Only *particular* shapes make sense, not *generic* ones
- If Shape is abstract, you can't create a new Shape
  - You *can* create a new Oval, a new Rectangle, etc.
  - Abstract classes are good for defining a general category containing specific, "concrete" classes

# An example abstract class

```
public abstract class Animal {
    abstract int eat();
    abstract void breathe();
}
```

- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the eat() and breathe() methods

# A problem

```
class Shape { ... }
class Star extends Shape {
    void draw() { ... }
    ...
}
class Circle extends Shape {
    void draw() { ... }
    ...
}
Shape s;
s = new Shape(); // Legal, but unwanted
s = new Star();  // Legal, because a Star is a Shape
s.draw();        // Illegal, Shape does not have draw()
```

# Another problem

```
Shape[] shapes = new Shape[16];
shapes[0] = new Circle();
shapes[1] = new Star();
…
for (Shape s : shapes) {
    s.draw(); // Illegal, Shape does not have draw()
}
```

# A solution

```
abstract class Shape {
    abstract void draw();
}
class Star extends Shape {
    void draw() { ... }
    ...
}
class Circle extends Shape {
    void draw() { ... }
    ...
}
Shape s;
s = new Shape(); // Illegal, Shape is abstract
s = new Star();  // Legal, because a Star is a Shape
s.draw();        // Legal, Shape does have draw()
```

# Interfaces

```
interface AffineT {
    public void move(double x, double y);
    public void rotate(double angle);
    public void scale(double scaleFactor);
}

interface SimilarT {
    public void generateSimilar();
}
```

- An interface declares methods but does not supply implementations
- All the methods are implicitly public and abstract. It may also contain constants (final attributes).
- Cannot be instantiated (An interface is like a *very* abstract class)

# Implementing an interface

- You extend a class, but you implement an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces

```
class Star extends Shape implements AffineT, SimilarT {
    public Star() {…}
    …
}
```

# Implementing an interface

```
class Star extends Shape implements AffineT, SimilarT {
    public Star() {…}

    public void move(double x, double y) {…}
    public void rotate(double angle) {…}
    public void scale(double scaleFactor) {…}

    public void generateSimilar() {…}
}
```

- When you say a class implements an interface, you are promising to *define* all the methods that were *declared* in the interface

# Partially implementing an Interface

- It is possible to define some but not all of the methods defined in an interface
  - Since this class does not supply all the methods it has promised, it is an abstract class
- It is possible to *extend* an interface (to add methods):
  - It is a new interface with additional methods

# What are interfaces for?

- A class can only extend one other class, but it can implement multiple interfaces
  - This lets the class fill multiple *roles*
  - In graphical interfaces (GUIs), it is common to have one class implementing several listeners (i.e., interfaces)
- Example:

```
class Application extends JFrame implements
ActionListener, KeyListener {
    ...
}
```

# Problem

```
public class GroudVehicle {
    activateWheels() {…}
    …
}

public class WaterVehicle {
    activateWaterFan() {…}
    …
}

// Not allowed in Java!! Only one class can be extended!
public class Anphibian extends GroudVehicle, WaterVehicle {
    …
}
```

# Solution

```
public interface GroudVehicle {
     activateWheels();
     …
}

public interface WaterVehicle {
     activateWaterFan();
     …
}

// OK!
public class Anphibian implements GroudVehicle, WaterVehicle {
     activateWheels() {…}
     activateWaterFan() {…}

     …
}
```

# Interfaces and instanceof

- instanceof is a keyword that tells you whether a variable "is a" member of a class or interface

```
class Dog extends Animal implements Pet {...}
Animal fido = new Dog();

fido instanceof Dog        //OK!
fido instanceof Animal     //OK!
fido instanceof Pet        //OK!
```

# Adapter classes

- When you implement an interface, you promise to define *all* the functions it declares
- There can be a *lot* of methods

```
interface KeyListener {
      public void keyPressed(KeyEvent e);
      public void keyReleased(KeyEvent e);
      public void keyTyped(KeyEvent e);
}
```

- What if you only care about a couple of these methods?

# Adapter classes

- Solution: use an adapter class
- An adapter class implements an interface and provides empty method bodies

```
class  KeyAdapter implements KeyListener {
     public void keyPressed(KeyEvent e) { };
     public void keyReleased(KeyEvent e) { };
     public void keyTyped(KeyEvent e) { };
}
```

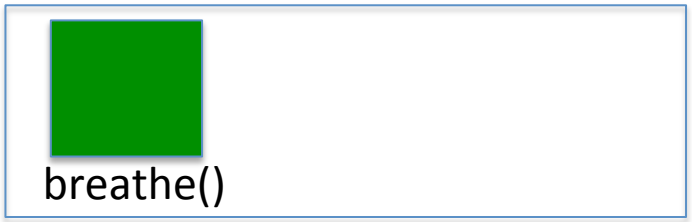
- You can override only the methods you care about. Java provides a number of adapter classes

# Summary

- We design a videogame allowing dogs to breathe, bark and move

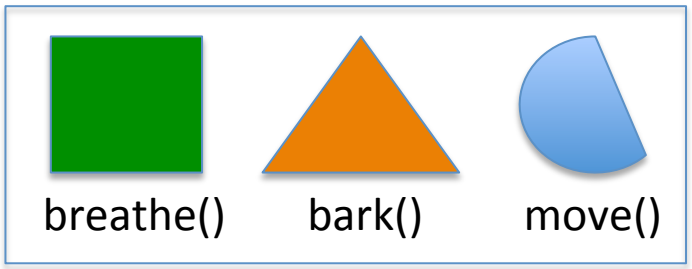
```
interface movable {
    public abstract void move(double x, double y);
}

class Animal {
    public void breathe();
}

class Dog extends Animal implements movable {
    public void bark();
}

Dog lessie = new Dog();
Animal a = lessie;
Movable m = lessie;
```

# Summary

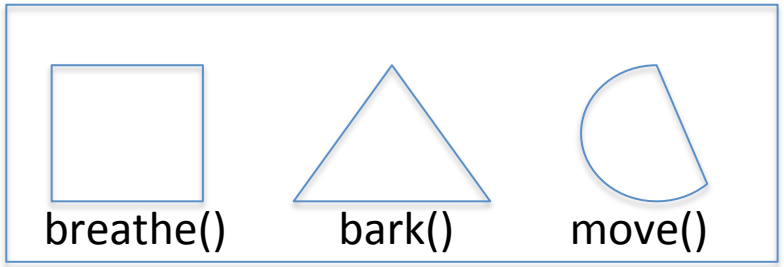



*a, the Animal reference*


*m, the movable reference*




*lessie, the Dog reference*




the actual Dog object

# Vocabulary

- abstract method—a method which is declared but not defined (it has no method body)
- abstract class—a class which either (1) contains abstract methods, or (2) has been declared abstract
- instantiate—to create an instance (object) of a class
- interface—similar to a class, but contains only abstract methods (and possibly constants)
- adapter class—a class that implements an interface but has only empty method bodies

*Complexity has nothing to do with intelligence, simplicity does.*

*— Larry Bossidy*
*Ex CEO Honeywell*

*Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away.*

*— Antoine de Saint Exupery*
*Scrittore, aviatore francese*