

# OOP Inheritance

---

Università di Modena e Reggio Emilia

*Prof. Nicola Bicocchi ([nicola.bicocchi@unimore.it](mailto:nicola.bicocchi@unimore.it))*



# Motivation

---

- 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
- A new design created by changing an existing design. (The new design consists of only the changes)
- Localization of code
  - Fixing a bug in the base class automatically fixes it in the subclasses
  - Adding functionalities to the base class automatically adds them to the subclasses
  - Reduced 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 attributes and methods of the class it inherited from**
- The inheriting class can **define additional attributes and methods**
- 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

# Example I (extension)

---

```
Class Car {  
    boolean isOn;  
    String licensePlate;  
  
    void turnOn() {...}  
    void turnOff() {...}  
}
```

```
class SDCar extends Car {  
    boolean isSelfDriving;  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```

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

*\*SD = Self Driving*

# Example II (override)

---

```
Class Car {  
    boolean isOn;  
    String licensePlate;  
  
    void turnOn() {...}  
    void turnOff() {...}  
}
```

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    /* method override */  
    void turnOn() {  
        turnSDOff();  
        /* ... */  
    }  
    /* method override */  
    void turnOff() {  
        turnSDOff();  
        /* ... */  
    }  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```



# Example III (override)

---

```
Class Car {  
    boolean isOn;  
    string licensePlate;  
  
    void turnOn() {...}  
    void turnOff() {...}  
}
```

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    /* override */  
    void turnOn() {  
        turnSDOff();  
        super.turnOn();  
    }  
  
    void turnOff() {  
        turnSDOff();  
        super.turnOff();  
    }  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```

# Class SDCar

---

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

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    /* override */  
    void turnOn() {  
        turnSDOff();  
        super.turnOn();  
    }  
  
    void turnOff() {  
        turnSDOff();  
        super.turnOff();  
    }  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}
```

# this and super

---

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

# 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 and Inheritance

---



# Recap

---

	Method in the same <b>class</b>	Method of another class in the same <b>package</b>	Method of <b>subclass</b>	Method of another public class in the outside <b>world</b>
<i>private</i>	✓			
<i>package</i>	✓	✓		
<i>protected</i>	✓	✓	✓	
<i>public</i>	✓	✓	✓	✓

# Visibility

---

```
Class Car {  
    private boolean isOn;  
    private string licensePlate;  
  
    public void turnOn() {...}  
    public void turnOff() {...}  
}
```

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

# Visibility

---

```
Class Car {  
    boolean isOn;  
    string licensePlate;  
  
    public void turnOn() {...}  
    public void turnOff() {...}  
}
```

```
Class SDCar extends Car {  
    void print() {  
        /*  
         * Works if Car and SDCar  
         * share the same package  
         */  
        System.out.println(licencePlate);  
    }  
}
```

# Visibility

---

```
Class Car {  
    protected boolean isOn;  
    protected string licensePlate;  
  
    public void turnOn() {...}  
    public void turnOff() {...}  
}
```

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

# Inheritance and constructors

---



# Construction of child objects

---

- Since each subclass “contains” an instance of the parent class, the latter **must be initialized**
- Java compiler automatically calls the **default constructor (no params!)** of the parent class
- The call is inserted as the **first statement** of each child constructor. If parent class disabled default constructor (by defining others) **parent constructor must be called explicitly!**

# super()

---

- **this** is a reference to the current object
  - **super** is a reference to the parent class
- 
- **super()** calls the default constructor of parent class
  - **super(params)** calls other constructors of parent class
    - Must be the first statement in child constructors

# Example

---

```
class Car {  
    boolean isOn;  
    String licensePlate;  
    /* Default constructor enabled! */  
}  
  
class SDCar extends Car {  
    boolean isSelfDriving;  
    /* Default constructor enabled! */  
} // Works!
```

# Example

---

```
class Car {  
    boolean isOn;  
    String licensePlate;  
    /* Default constructor enabled! */  
}  
  
class SDCar extends Car {  
    boolean isSelfDriving;  
    /* Default constructor disabled! */  
    public SDCar() {  
        /* automatic call to parent default constructor here! */  
    }  
} // Works!
```



# Example

---

```
class Car {  
    boolean isOn;  
    String licensePlate;  
    /* Default constructor disabled! */  
    public Car(String licensePlate) {  
        this.licencePlate = licensePlate;  
    }  
}  
  
class SDCar extends Car {  
    boolean isSelfDriving;  
    /* Default constructor disabled! */  
    public SDCar() {  
        /* automatic call to parent default constructor here! */  
    }  
} // Not working!
```



# Example

---

```
class Car {  
    boolean isOn;  
    String licensePlate;  
    /* Default constructor disabled! */  
    public Car(String licensePlate) {  
        this.licencePlate = licensePlate;  
    }  
}  
  
class SDCar extends Car {  
    boolean isSelfDriving;  
    /* Default constructor disabled! */  
    public SDCar(String licencePlate) {  
        super(licencePlate);  
        isSelfDriving = false;  
    }  
} // Works!
```



# Construction of child objects

---

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

# Example

---

```
class Car{  
    Car() {  
        super();  
        System.out.println("New Car created");  
    }  
}  
Class SDCar extends Car {  
    SDCar() {  
        super();  
        System.out.println("New SDCar created");  
    }  
}  
  
SDCar c = new SDCar(); // 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  
     * is not knowable at compile time! */  
}
```



# 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 (**polymorphism**) depending on the actual class of the object.

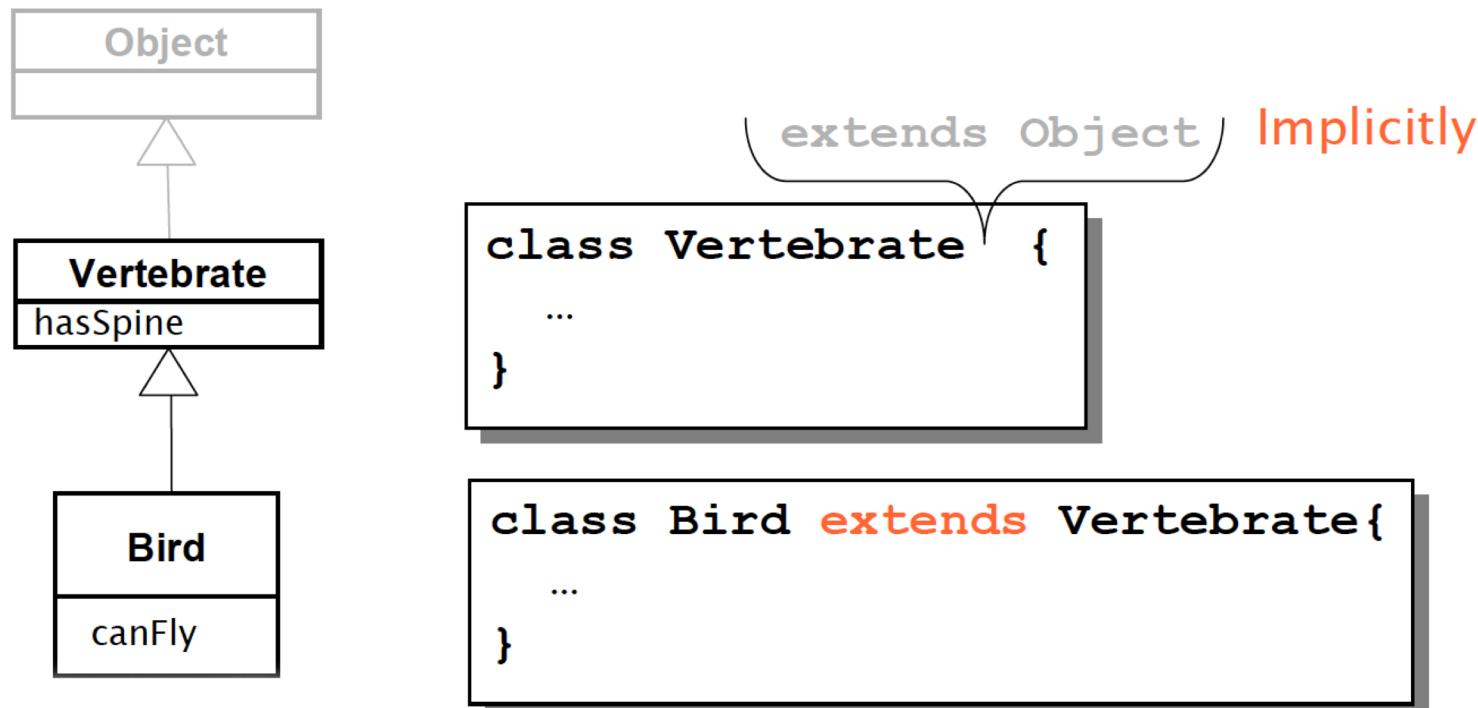
\* [https://en.wikipedia.org/wiki/Late\\_binding](https://en.wikipedia.org/wiki/Late_binding)

# Object

---

# Java.lang.Object

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



# Java.lang.Object

---

- All objects can be seen as Object instances
- Object defines basic services, which are useful for all classes. They are often overridden in sub-classes. For example:
  - `toString()`: returns a string representation
  - `equals(Object o)`: tests equality
  - `clone()`: returns a shallow copy of the object

# toString()

---

```
class Car{  
    String licencePlate;  
    public String toString(){  
        return “[Car] ” + licencePlate;  
    }  
}  
  
Car c = new Car();  
// println(Object) call  
System.out.println(c);  
  
// println(String) call  
System.out.println(c.toString());
```



# equals(Object o)

---

```
public class Car {  
    boolean isOn;  
    String licencePlate;  
    public Car(boolean isOn, String licencePlate) {  
        this.isOn = isOn;  
        this.licencePlate = licencePlate;  
    }  
    public static void main(String[] args) {  
        Car c1 = new Car(true, "AA334GG");  
        Car c2 = new Car(true, "AA334GG");  
        System.out.println(c1.equals(c2)); //false!  
    }  
}
```



# equals(Object o)

---

```
public class Car {  
    boolean isOn;  
    String licencePlate;  
    public Car(boolean isOn, String licencePlate) {  
        this.isOn = isOn;  
        this.licencePlate = licencePlate;  
    }  
  
    /* the equals() methods must be overridden */  
    public boolean equals(Object obj) {  
        Car other = (Car) obj;  
        if (isOn != other.isOn)  
            return false;  
        if (licencePlate != other.licencePlate)  
            return false;  
        return true;  
    }  
}
```



# Casting

---



# Types

---

- Java is a strictly typed language. Each variable has a type!

```
float f;  
f = 4.7;           //OK!  
f = "hello!";     //!OK
```

```
Car c;  
c = new Car();    //OK!  
c = new String(); //!OK
```



# Upcasting and Downcasting

---

```
class Car {};  
class SDCar extends Car {};
```

```
Car c1 = new Car();      // OK!  
SDcar c2 = new SDCar (); // OK!
```

But also...

```
Car c3 = new SDcar();
```

# Upcasting and Downcasting

---

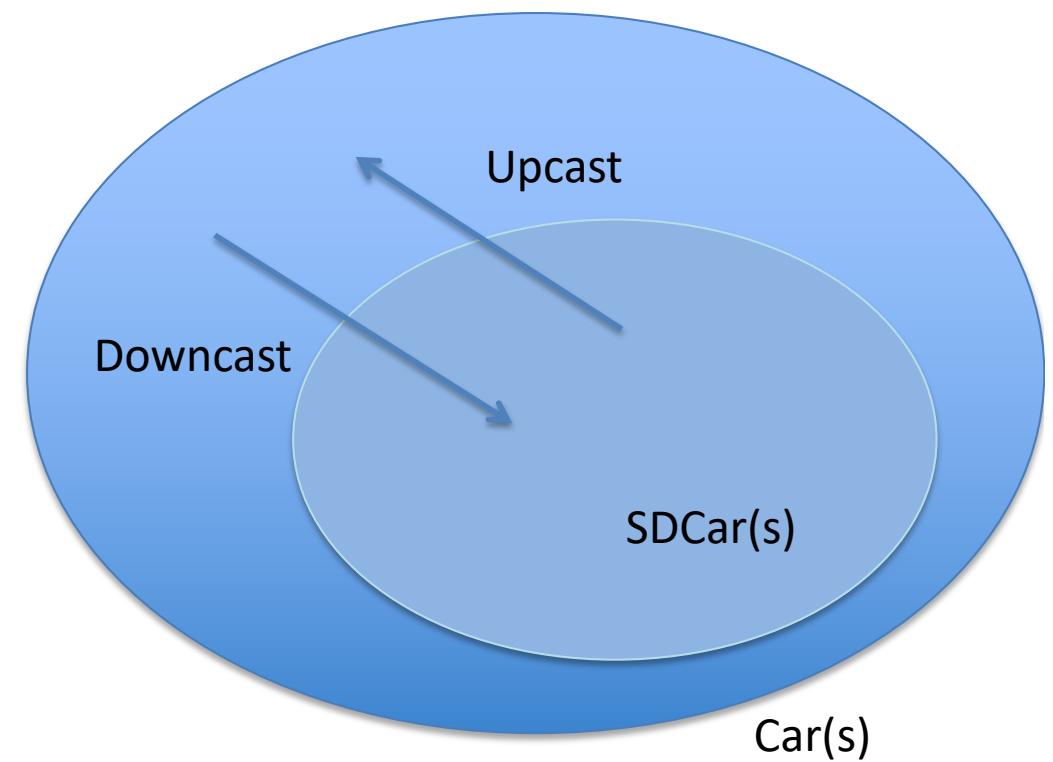
```
Car c3 = new SDcar();
```

Specialization defines a sub-typing relationship (**is a**).

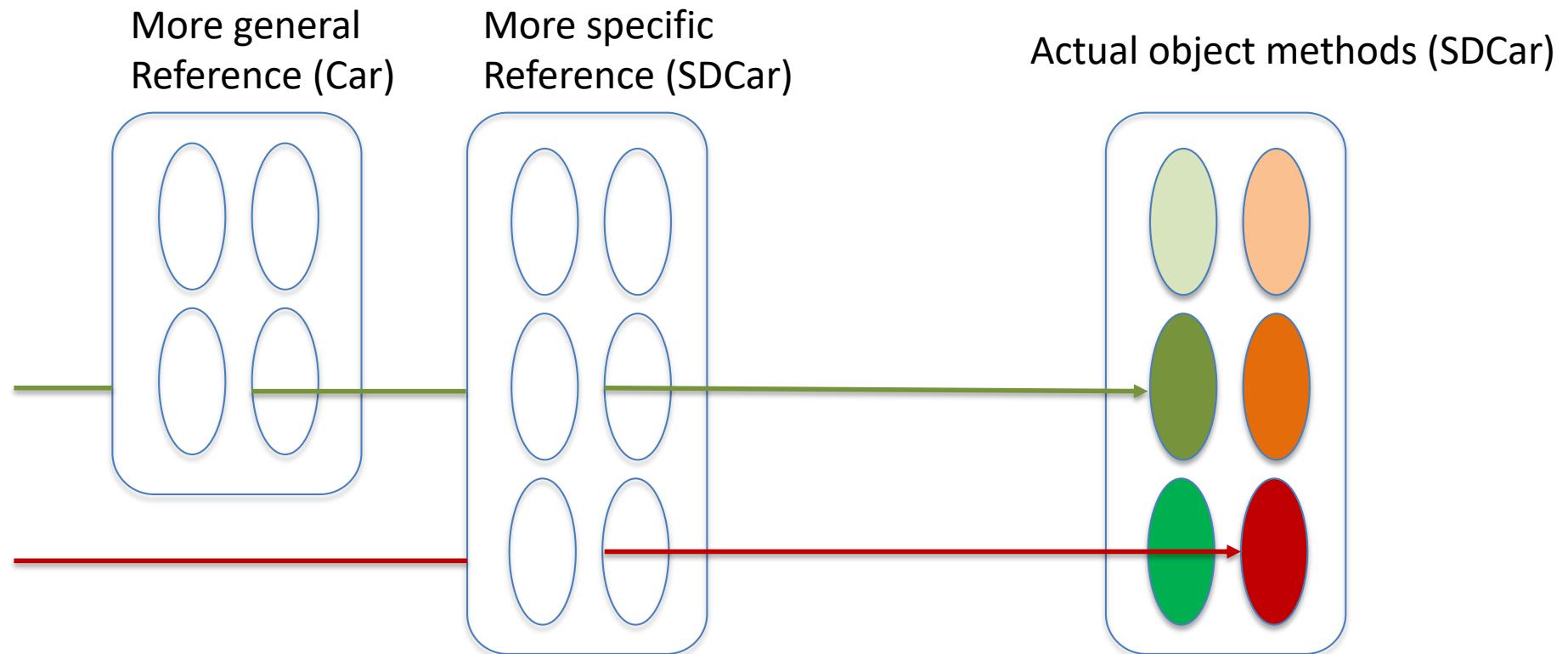
All SDCar(s) are Car(s). Not all Car(s) are SDCar(s).

Upcasting and downcasting refer to the possibility of changing the reference type of a given object.

Upcasting consists in using more general references, while downcasting more specific references.



# Upcasting and Downcasting



# Upcasting

---

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

- Assignment from a more specific type to a more general type
- Note well: reference type and object type are separate concepts. Object referenced by 'c' continues to be of SDCar type! Only the interface changes!

# Upcasting

---

- It is **dependable**
  - It is always true that an SDCar is a Car too
- It is **automatic**
  - `Car c = new SDCar();`

# Example

---

```
Class Car {  
    boolean isOn;  
    string licensePlate;  
  
    void turnOn() {...}  
    void turnOff() {...}  
}
```

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

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```

```
Car c2 = c1; // Upcast  
c2.turnSDOn() // Compile time error!  
(Car 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() {...}  
}
```

```
Car c1 = new SDCar();  
c1.turnSDOn() // Compile time error!
```

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```

```
SDCar c2 = (SDcar)c1; // Downcast  
c2.turnSDOn() // Accidentally OK! The object  
referenced by c1 was actually of class SDCar
```



# Example

---

```
Class Car {  
    boolean isOn;  
    string licensePlate;  
  
    void turnOn() {...}  
    void turnOff() {...}  
}
```

```
Car c1 = new Car();  
c1.turnSDOn() // Compile time error!
```

```
Class SDCar extends Car {  
    boolean isSelfDriving;  
  
    void turnSDOn() {...}  
    void turnSDOff() {...}  
}
```

```
SDCar c2 = (SDcar)c1; // Downcast  
c2.turnSDOn() // Run time error!
```



# 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 SDCar();
if (c instanceof SDCar){
    SDCar sdc = (SDCar) c;
    sdc.turnSDOn();
```

```
}
```



# 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 = (Object)any;
```

# Abstract Classes and Interfaces

---



# Abstract methods

---

- You can *declare* an *object* without *implementing* it:  
Person p;
- Similarly, you can *declare* a *method* without *implementing* it (i.e., the body of the method is missing):  
`public abstract void draw(int size);`
- A method that has been **declared** but not **defined** is an **abstract method**

# Abstract classes

---

- Any class containing **one or more abstract methods** is an **abstract class**
- An abstract class is *incomplete* in the sense it has missing method bodies
- You must declare the class with the keyword **abstract**:

```
abstract class MyClass { . . . }
```
- 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 just prevents the class from being instantiated

# Why use abstract classes?

---

- Suppose you wanted to create a class **Shape**, with subclasses **Oval**, **Rectangle**, **Triangle**, **Hexagon**, etc.
- Each subclass has a method **draw()** for representing its shape on a 2D graphic panel

# 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.draw();        // Illegal, Shape does not have draw()
s = new Star(); // Legal, because a Star is a Shape
s.draw();        // Illegal, Shape does not have draw()
```



# Same problem, another view

---

```
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()
```



# Another problem

---

- Let's suppose that all shapes must have two capabilities:
  - Drawing their own shape [draw() method]
  - Setting a unique ID [setID() method]

# A solution

---

We can keep the Shape class abstract while providing a common implementation of the setID() method.

## Benefits

1. Shape cannot be instantiated
2. Shape subclasses can redefine draw()

## Drawbacks

1. We partially lose the possibility to define abstract concepts such as Shape. Now Shape contains code!

```
abstract class Shape {  
    void setID() { . . . };  
    abstract void draw();  
}  
  
class Star extends Shape {  
    void draw() { . . . }  
    ...  
}  
  
class Circle extends Shape {  
    void draw() { . . . }  
    ...  
}
```



# A better solution

We can use a Shape **Interface**.

- Interfaces are special classes for declaring methods without supplying implementations
- All their methods are implicitly public and abstract
- Interfaces cannot be instantiated because they do not contain actual code
- When a class implements an interface, it promises to *define* all the methods *declared* in the interface

```
interface Shape() {  
    public abstract void setID();  
    public abstract void draw();  
}  
  
abstract class AbstractShape implements  
Shape {  
    void setID() { . . . };  
    abstract void draw();  
}  
  
class Star extends AbstractShape {  
    void draw() { ... }  
    ...  
}  
  
class Circle extends AbstractShape {  
    void draw() { ... }  
    ...  
}
```

Level of abstraction

# Specialization and partial implementation

---

- Interfaces can be specialized
  - Specializing an interface means adding new methods in derived interfaces. Overriding methods does not make sense in interfaces because code is absent.
- Interfaces can be partially implemented
  - Partial implementations of interfaces can be found in abstract classes. The unimplemented methods must be marked as abstract.

# Multiple inheritance

---

- In Java, a class can only extend one class, but can implement multiple interfaces
  - This lets the class fill multiple *roles* (i.e., multiple set of methods)
  - In graphical interfaces (GUIs), it is common to have one class implementing several listeners (i.e., interfaces)

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



# Multiple inheritance

---

```
public class GroudVehicle {  
    activateWheels() {...}  
  
    ...  
}  
public class WaterVehicle {  
    activateWaterFans() {...}  
  
    ...  
}  
// Not allowed in Java!! Only one class can be extended!  
public class Amphibian extends GroudVehicle, WaterVehicle {  
  
    ...  
}
```



# Multiple inheritance

---

```
public interface GroudVehicle {  
    activateWheels();  
    ...  
}  
public interface WaterVehicle {  
    activateWaterFans();  
    ...  
}  
public class Amphibian implements GroudVehicle, WaterVehicle {  
    activateWheels() {...}  
    activateWaterFans() {...}  
    ...  
}
```



# Interfaces and instanceof

---

- **instanceof** is a keyword that tells you whether a variable “is a” member of a class or interface
- Membership of a class or interfaces can be translated with “**has its methods implemented**”

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

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

# 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*