Inheritance

Vũ Thị Hồng Nhạn

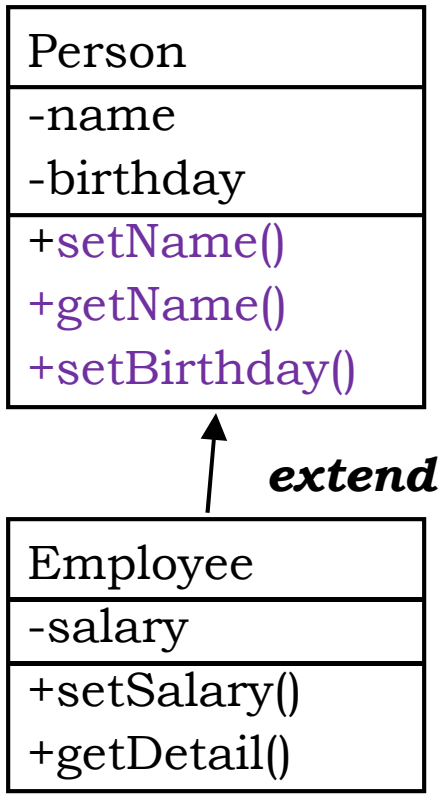
(vthnhan@vnu.edu.vn)

Dept. of Software Engineering, UET

Vietnam National Univ., Hanoi

Access modifiers in Java

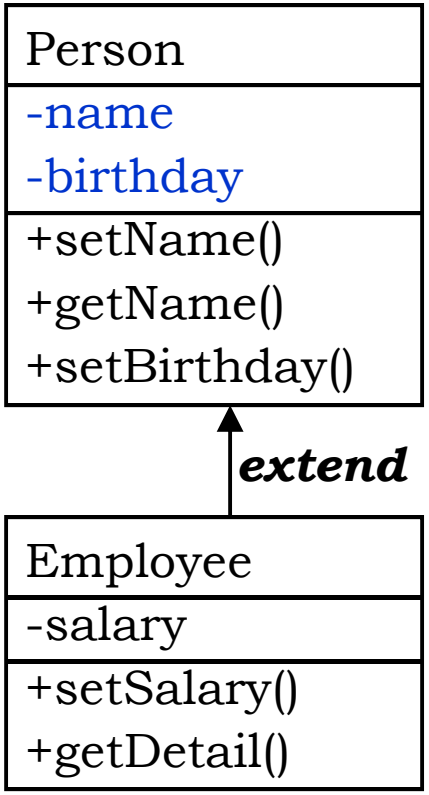
Example




```
Employee e = new Employee();
...
e.setName("John");
e.setSalary(3.0);
System.out.print(e.getName());
```

Accessing members of the base class from subclass

```
class Employee extends Person {
    ...
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        //s = getName() + "," + getBirthday();
        s += "," + salary;
        return s;
    }
}
```



Access modifiers

- Help restrict *the scope* of *a class, constructor, variable, method,* or *data member*
- Four types of access modifiers in java
  1. **default:** no keyword required
  2. **private**
  3. **protected**
  4. **public**

Example

in the same package

```
public class Person {
    Date birthday;
    String name;
...
}
```

default

```
public class Employee extends Person {
...
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

The **protected** access modifier

- The *methods* or *data members* declared as **protected** are accessible **within …**
  - **same package**
  - or **sub classes** in different packages

Example: in same package

```
public class Person {
    protected Date birthday;
    protected String name;
…
}

public class Employee extends Person {
…
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

Example

in different packages

```
package abc;
public class Person {
    protected Date birthday;
    protected String name;
…
}
```

```
import abc.Person;
public class Employee extends Person {
…
    public String getDetail() {
        String s;
        s = name + "," + birthday;
        s += "," + salary;
        return s;
    }
}
```

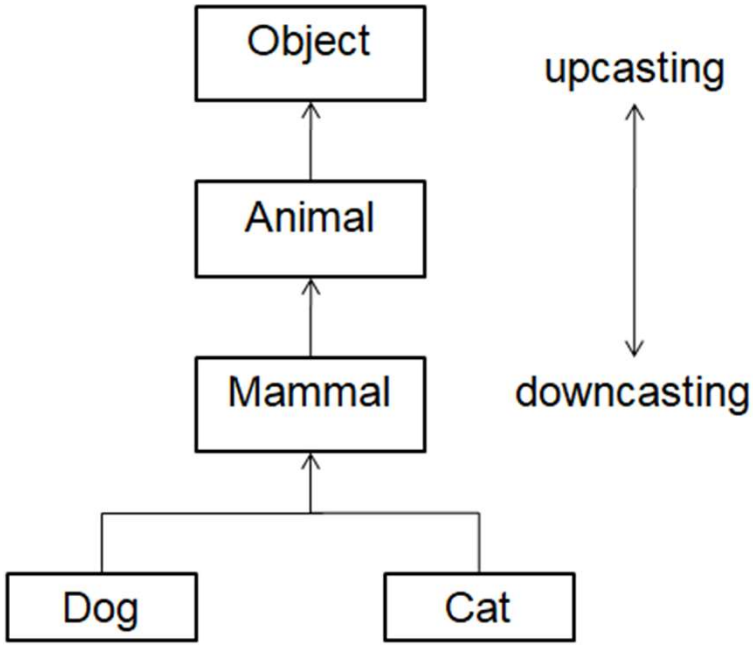
4 types of access modifiers...

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Casting objects

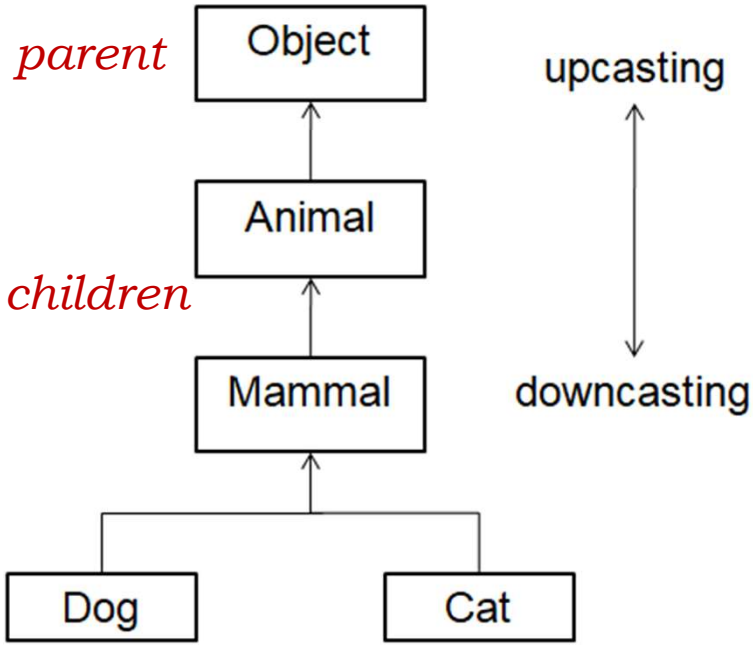
Upcasting vs. Downcasting

- Java allows an object of a subclass to be treated as an object of any super class
  - This is called upcasting
  - **Upcasting is done automatically**
- **Downcasting** is also allowed but it must be **done manually**
- But upcasting & downcasting are not like casting primitive data types



Upcasting

- **Cat** & **Dog** are both Mammals, which extends from Animal, which *automatically extends* from Object
- in Java, everything is an Object except **primitives**
- Is **a Cat** an Object?
  - Yes, because by inheritance **Cat** gets all the properties its ancestors have
  - Cat is also an Animal and a Mammal too

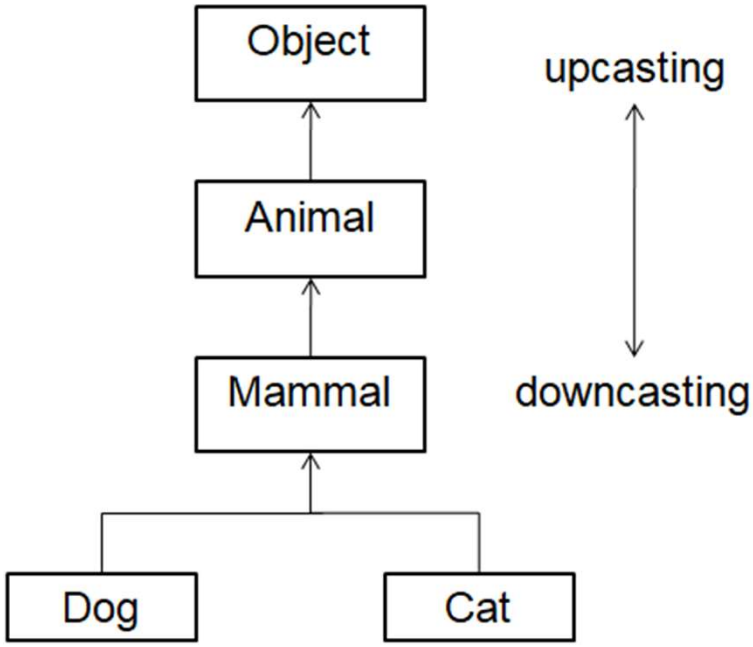


Animal hierarchy

Upcasting: example

```
class Animal {
    int health = 100;
}
class Mammal extends Animal {}
class Cat extends Mammal {}
class Dog extends Mammal {}

public class Test {
    public static void main(String[] args) {
    Cat c = new Cat();
    System.out.println(c.health);
    Dog d = new Dog();
    System.out.println(d.health);
     }
}
```

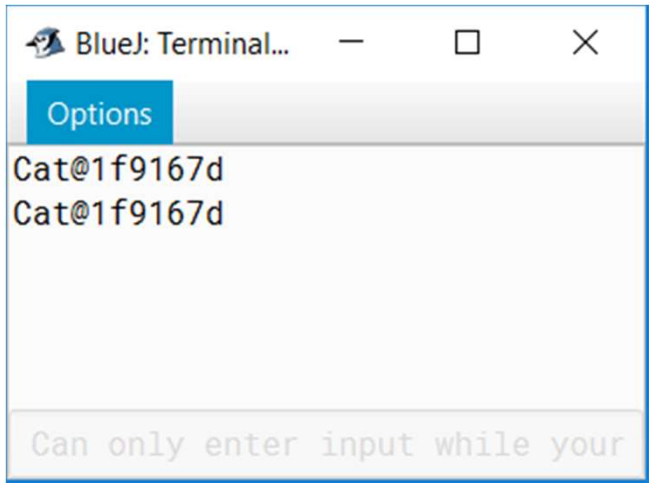


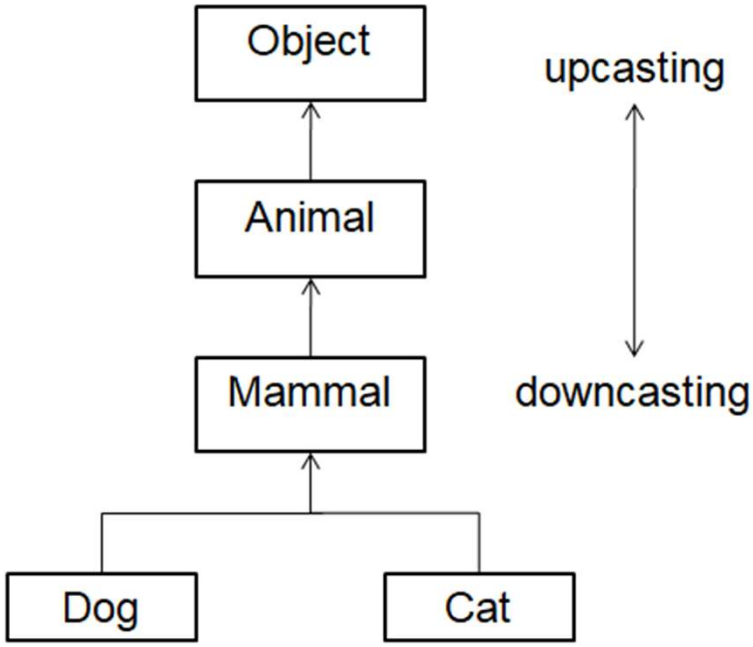
Output?

Upcasting: example

```
Cat c = new Cat();
 System.out.println(c);
Mammal m = c; // upcasting
System.out.println(m);
```

Output?





→ *Cat is exactly the same after casting . Cat didn't change to a Mammal, it is just being labeled Mamal now*

Upcasting…

- Upcasting can be done automatically
- E.g.,

```
Mammal m= (Mammal) new Cat();
```

is equal to

```
Mammal m = new Cat();
```

Downcasting

- Downcasting must always be done manually

- Cat c1 = **new** Cat();
- Animal a= c1; *//upcasting to an Animal automatically*
- Cat c2= (Cat) a; *//manually downcasting back to a Cat*

instanceof

- is used to test if an object is an instance of some class

```
Cat c1 = new Cat();
Animal a = c1; //upcasting to Animal
if(a instanceof Cat){ // testing if the Animal is a Cat
System.out.println("It's a Cat! safely downcast it to a Cat");
Cat c2 = (Cat)a;
```

- Don't confuse **variables** with **instances**!
  - Cat from a Mammal variable can be cast to a Cat
  - but, Mammal from a Mammal variable cannot be cast to a Cat

Casting…

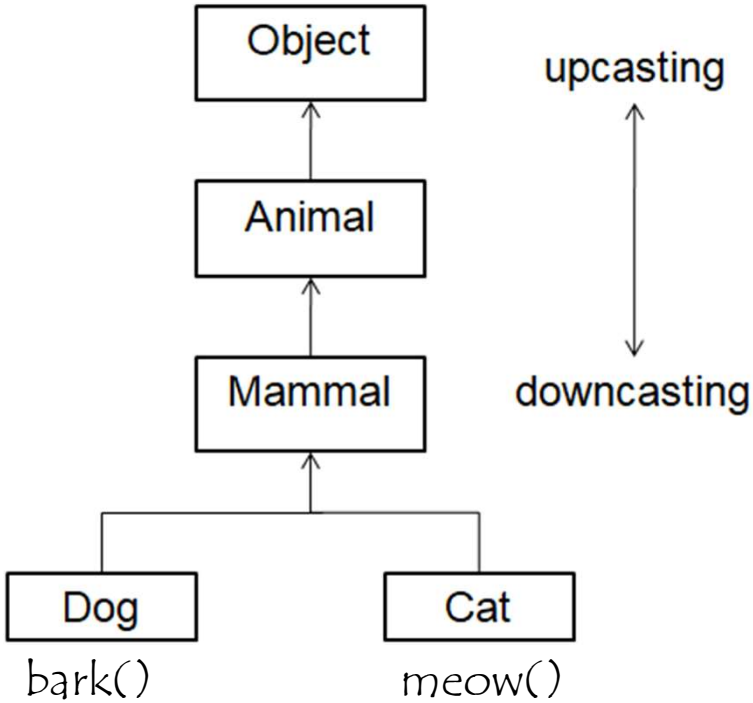
- Casting cannot always be done in both ways
- if you're creating a Mammal (by calling new Mammal())
  - but it cannot be downcasted to Dog or Cat
- E.g.

```
Mammal m = new Mammal();
Cat c = (Cat)m;
```

  - This code passes compiling, but throw java.lang.ClassCastException exception while running because Mammal is not a Cat but we're trying to cast to a Cast

Casting…

- If you upcast an object, it will lose all the properties which were inherited from its current position
- Data will not be lost, you just can't use it till you downcast the object to the right level
- Why?
  - If you have a group of animals, then you cannot be sure which ones can meow() and which ones can bark()
  - That's why you cannot make animal do things!



Upcasting during method calling

- We can make general methods, which can take different classes as an argument

```
public static void stroke(Animal a){
    System.out.println("you stroke the " + a);
}

Cat c= new Cat();
Dog d= new Dog();

stroke(c);
stroke(d);

Animal aa=c;
stroke(aa);
```

- What about…
  - c.meow()
  - aa.meow()

Must downcasting manually➔ **((Cat)aa).meow()**