



# Relationships in UML Class Diagrams

---

Some examples using `Java`



# Exercise: draw the UML

---

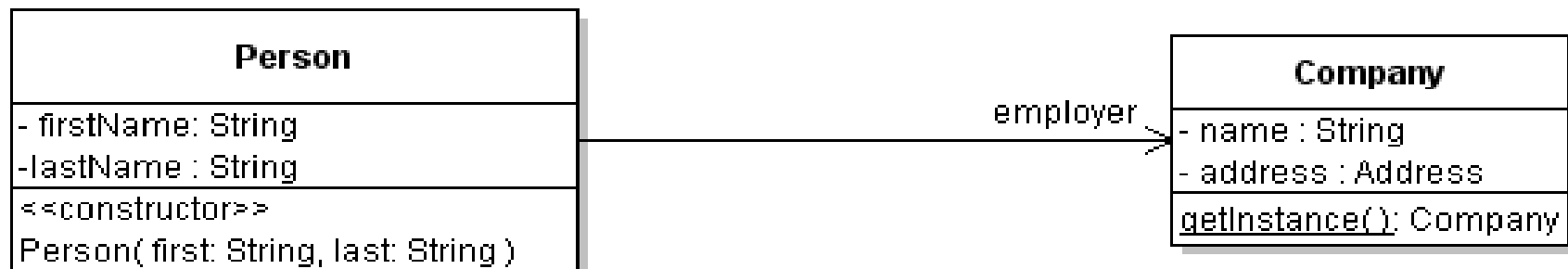
A Person works for one company.

```
class Person {  
    private String firstName;  
    private String lastName;  
    private Company employer;  
    ...  
}
```

# Unidirectional Association

A Person works for one company.

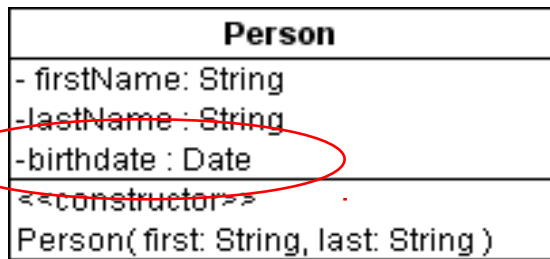
```
class Person {  
    private String firstName;  
    private String lastName;  
    private Company employer;  
    ...  
}
```



# Association or Attribute?

Should **birthdate** be an **attribute** or **association** to the Date class?

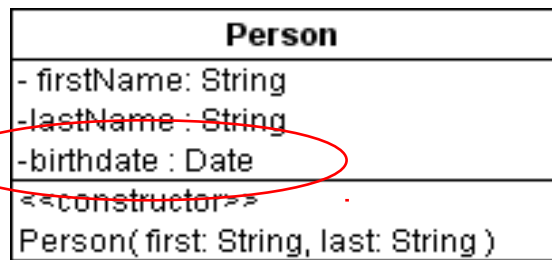
```
class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthdate;  
    private Company employer;  
    ...  
}
```



# Association or Attribute?

For **uninteresting** objects like String and Date, show as attribute.

```
class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthdate;  
    private Company employer;  
    ...  
}
```





# Multiplicity of Association

---

A person works for 0 or 1 company, a company has many employees.

```
class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthdate;  
    private Company employer; // may be null  
    ...  
}
```

# Multiplicity of Association

A person works for 0 or 1 company, a company has many employees.

```
class Person {  
    private String firstName;  
    private String lastName;  
    private Date birthdate;  
    private Company employer;  
    ...  
}
```





# Bidirectional Association

---

A company has a collection (e.g. Set) of employees (no duplicates).

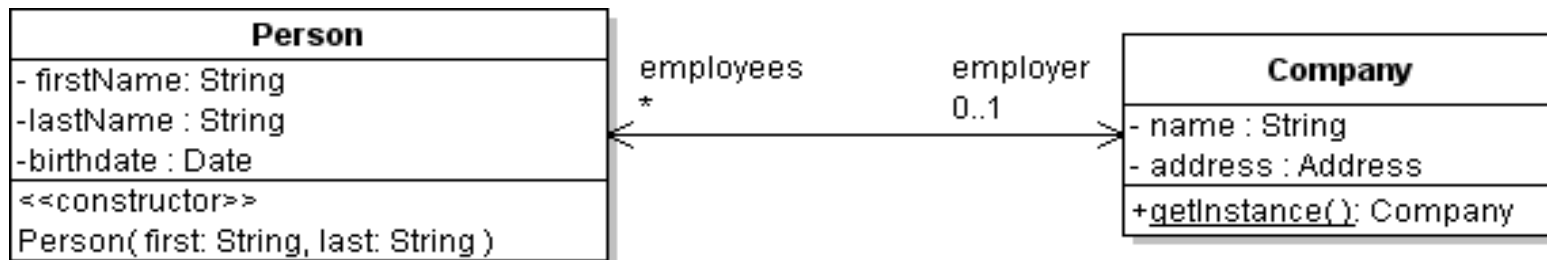
```
class Company {  
    private String name;  
    private Address address;  
    private Set<Person> employees;  
    public static Company getInstance( );  
}
```



# Bidirectional Association

A company has a collection (e.g. Set) of employees (no duplicates).

```
class Company {  
    private String name;  
    private Address address;  
    private Set<Person> employees;  
    public static Company getInstance( );  
}
```





# Composition: owning a collection

---

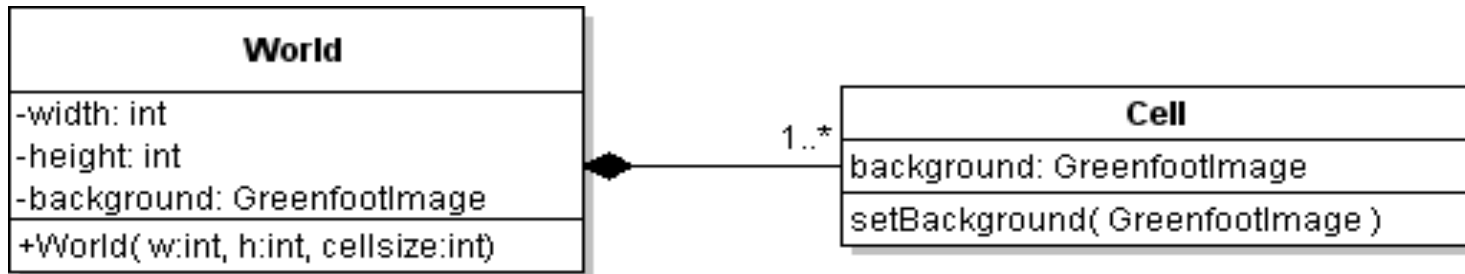
A Greenfoot World "owns" the cells in the world.

```
public class World {  
    private [][] Cell cells;  
    public World(int width, int height, int sz)  
    {  
        if (width<1 || height<1) throw new ...;  
        cells = new Cell[width][height];  
    }  
}
```

# Composition: owning a collection

A Greenfoot World "owns" the cells in the world.

```
public class World {  
    private [][] Cell cells;  
    public World(int width, int height, int sz)  
    {  
        if (width<1 || height<1) throw new ...;  
        cells = new Cell[width][height];  
    }  
}
```





# Collection can be array, list, set, ...

A greenfoot World "owns" the actors in the world.

```
public class World {  
    private List<Actor> actors;  
    public void addObject(Actor a, int x, int y)  
    {  
        if ( ! actors.contains(a) ) actors.add(a);  
        else actors.add( a ); a.setLocation(x,y);  
    }  
}
```

Exercise: draw the class diagram



# Association to what?

---

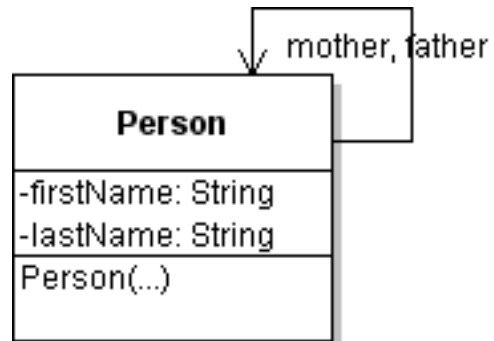
A person has a mother and father.

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Person father;  
    private Person mother;  
    ...  
}
```

# Self-Association

A person has a mother and father.

```
public class Person {  
    private String firstName;  
    private String lastName;  
    private Person father;  
    private Person mother;  
    ...  
}
```





# Implementing an Interface

---

A Person can be compared to another Person.

```
public class Person implements Comparable<Person> {  
    private String firstName;  
    private String lastName;  
    private Date birthdate;  
  
    public int compareTo(Person other) {  
        if (other == null) return -1;  
        ...  
    }  
}
```



# Key Words and Stereotypes

---

Convey extra information about a class, an attribute, or a relationship.

## Stereotypes

<<abstract>>

<<interface>> **or** <<I>>

<<enum>>

**for relationships:**

<<use>>      **this is useless**

<<create>>

## Key Words

{abstract}

{readonly}

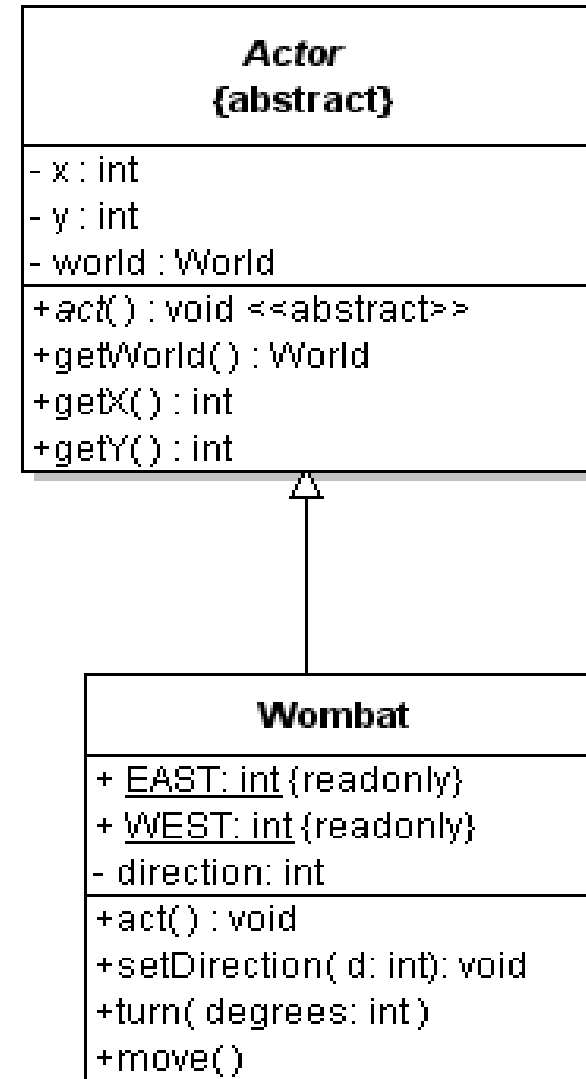
{ordered}

{unique}

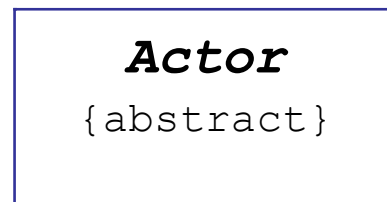
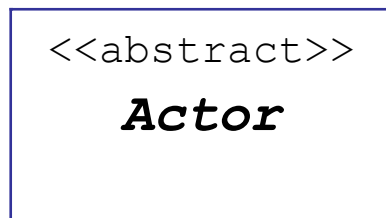


# Stereotypes and Qualifiers

- ***Actor*** is an abstract class
  - name in *Bold-Italic*
- *act( )* is abstract method
  - name in *italic font*
- EAST and WEST are class constants



Two ways to show abstract class:





# Dependency

---

- **Association** implies an *attribute* of an object
- **Dependency** just means one class somehow requires another class:
  - **type of parameter** to a method
  - **creates** a local object of the other class
  - **calls** a method of another class



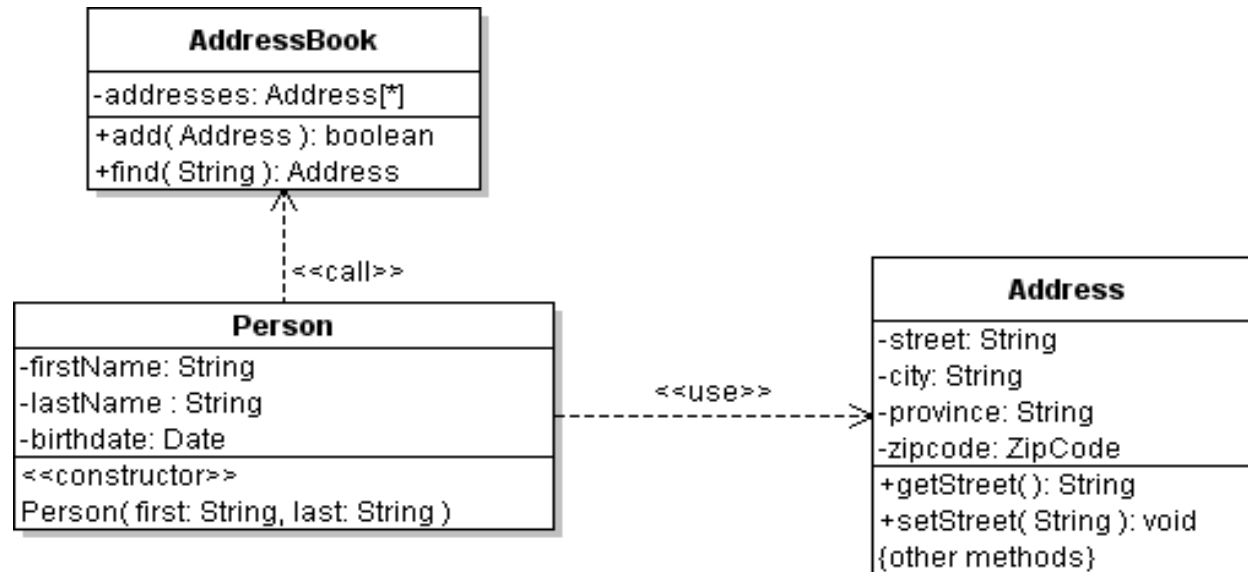
# Uses

---

```
public class Person {  
    findAddress( AddressBook book, String what){  
        Address address = book.find( what );  
        ...  
    }  
}
```

# Uses

```
public class Person {  
    findAddress( AddressBook book, String what){  
        Address address = book.find( what );  
        ...  
    }  
}
```





# Inheritance

---

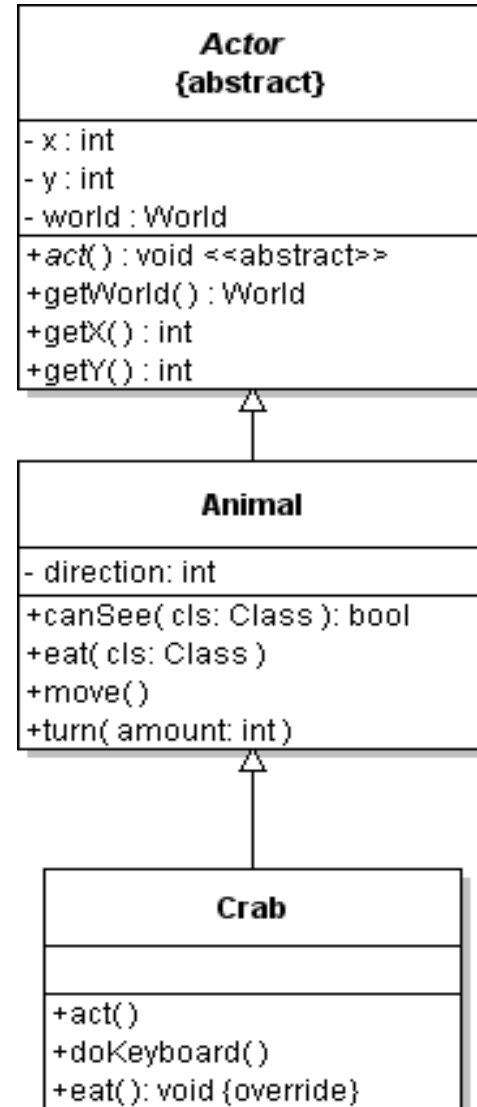
In Greenfoot...

- Animal is a **kind of** Actor
- Crab is a **kind of** Animal

```
public abstract class Actor {
    public abstract void act( );
}
public class Animal extends Actor {
    public void act( ) { /* do nothing */ }
}
public class Crab extends Animal {
    public void act( ) { /* move and eat worms */
        if (canSee( Worm.class ) )
            eat(Worm.class);
        ...
    }
}
```

# Inheritance or Generalization

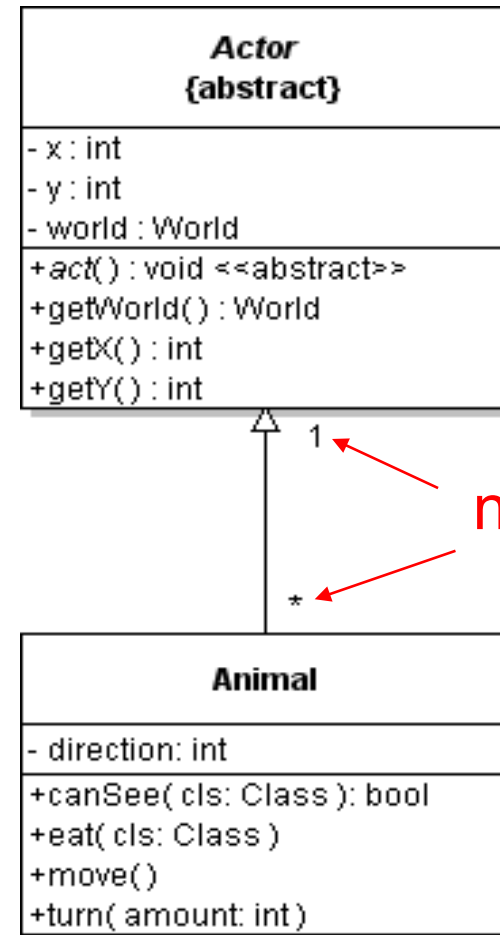
- Crab is a **kind of** Animal
- Animal is a **kind of** Actor
- in Java and C#, a class can only extend **one** other class



# No multiplicity on inheritance/depends

multiplicity is used **only** on associations, **not**

- inheritance
- dependency
- implements



# Implementing an Interface

- An interface *specifies* a type of behavior, but no implementation
- Methods in an interface are automatically *public* and *abstract*
- in Java, a class can implement **any number** of interfaces

