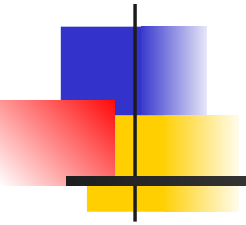
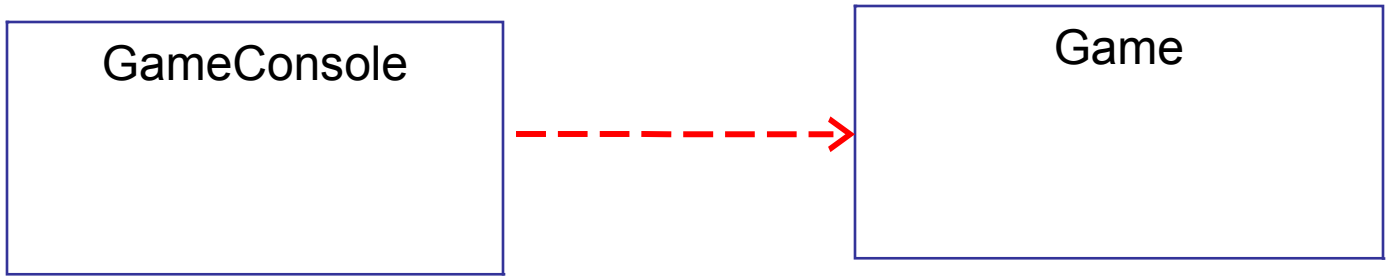


Showing Relationships in UML

Class Diagram with more than one class

Dependency

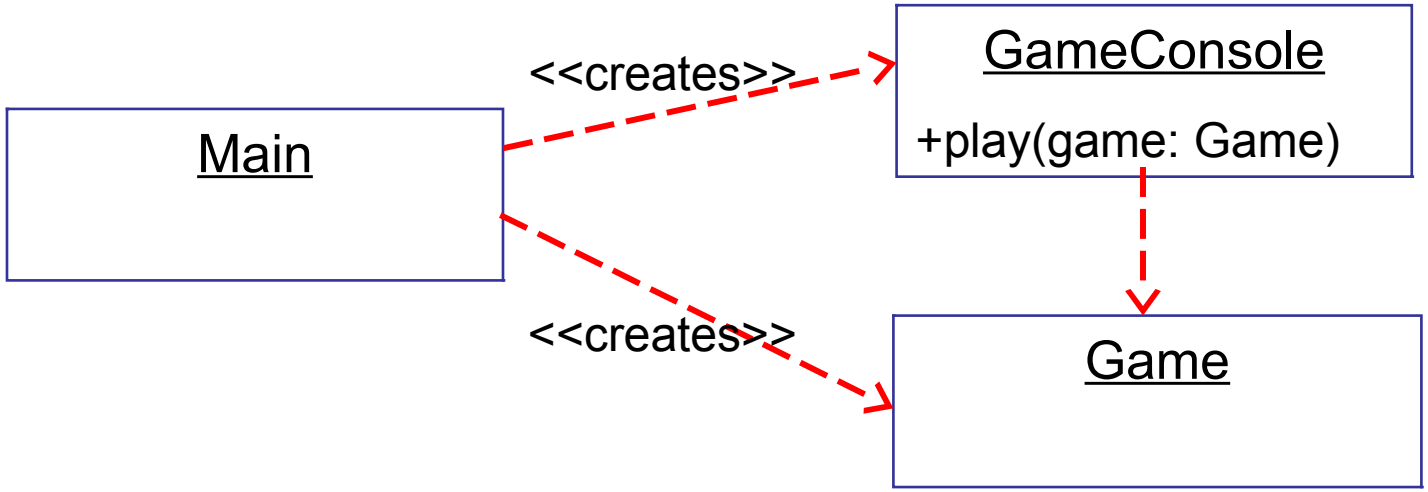
- One class uses or depends on another class.
- Includes "association".




```
public class GameConsole {
   // the play method depends on Game.
   public void play(Game game) {
      ...
      boolean correct = game.guess( number );
      String hint = game.getMessage( );
```

More Dependency

- Main depends on (uses) Game and GameConsole



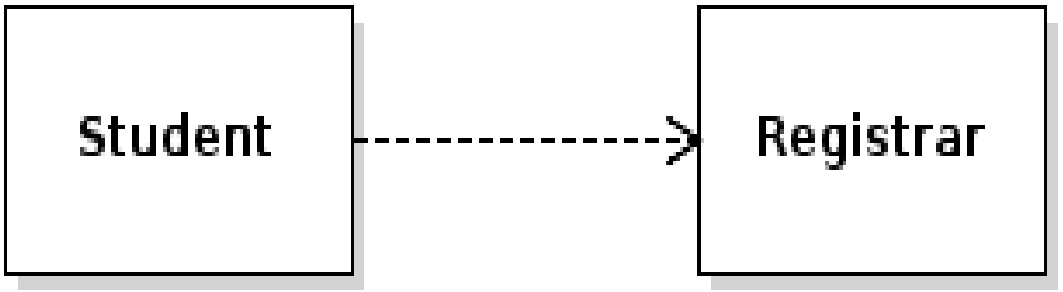

```
public class Main {
   public static void main(String[] args) {
      Game game = new Game(1000000);
      GameConsole ui = new GameConsole();
      ui.play( game );
```

Dependency Example

A Student *uses* the Registrar to enroll in a Course, but he doesn't save a reference to the Registrar.

```
public class Student {
    private long id;
    //NO Registrar attribute!

    public void addCourse(Course course) {
        Registrar regis = Registrar.getInstance();
        regis.enroll(this, course);
```



Association

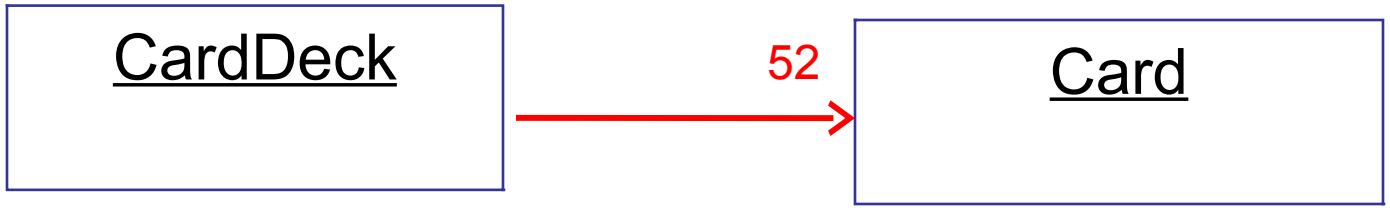
- Association means one object has an attribute of another class.


```
public class GameConsole {
   private Game game;
   /** constructor */
   public GameConsole(Game game) {
      this.game = game;
```

Association with Multiplicity

- You can indicate *multiplicity* of the association.

A card deck contains exactly 52 cards.




```
public class CardDeck {
   private Card[] cards;
   public CardDeck() {
      cards = new Card[52];
      for(int k=0; k<52; k++) {
         cards[k] = new Card("...");
      }
```

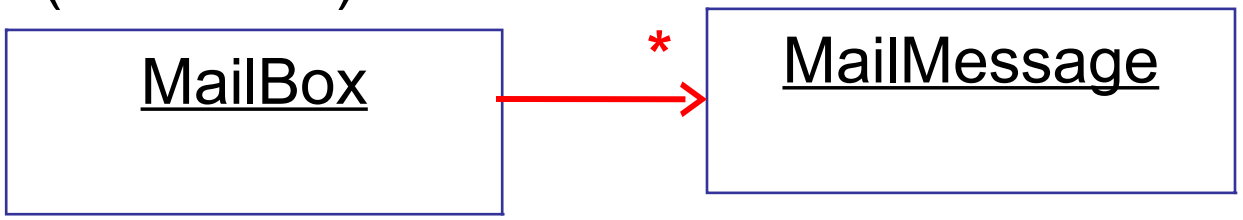
Association with Variable Multiplicity

A MailBox may contain 0 or more mail messages.

* = any number (0 or more)

1..n = 1 to n

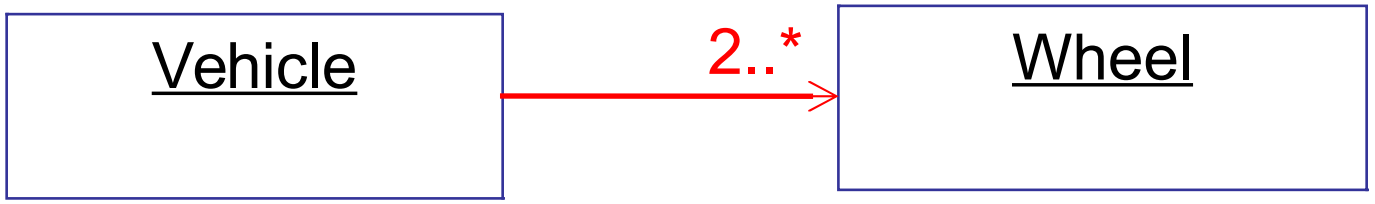
n = exactly n




```
public class MailBox {
   private List<MailMessage> messages;
   public MailBox() {
      messages = new ArrayList<MailMessage>();
   }
   public void addMessage(MailMessage m) {
      message.add( m );
   }
```

Vehicle has at least 2 Wheels

Only vehicles with at least 2 wheels are allowed on roads. A vehicle must have at least 2 wheels.

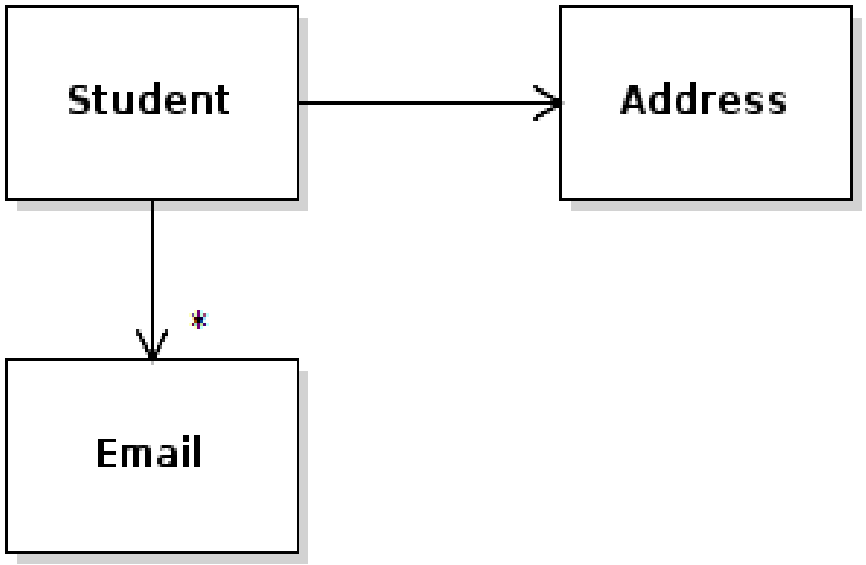



```
public class Vehicle {
   private Wheel[] wheels;
   public Vehicle(int n) {
      if (n<2) throw new IllegalArgumentException(
         "Must have at least 2 wheels.");
      wheels = new Wheel[n];
   }
```

Class with Many Associations

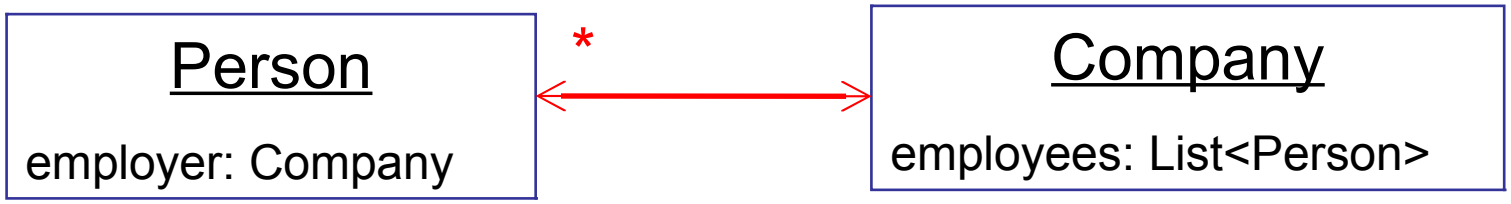
A Student *has* one physical Address and 0 or more Email address.

```
public class Student {
    private Address homeAddress;
    /** his email addresses. He may have many. */
    private List<Email> emailAddress;
```



Bidirectional Association.

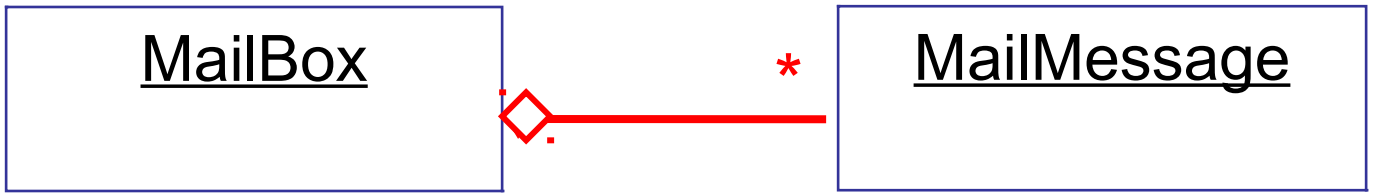
If each object has a *reference* to the other object, then it is *bidirectional.*



This is rare, in practice.
Try to avoid bidirectional associations.

Aggregation: whole-parts relationship

One class "collects" or "contains" objects of another




```
public class MailBox {
   private List<MailMessage> messages;
   /* a MailBox consists of MailMessages */
```

Aggregation often shows a whole-parts relationship

The parts *can exist* without the whole. (MailMessage can exist outside of a MailBox.)

When to use Aggregation?

- One object "collects" or "aggregates" components.

Advice: **Don't show aggregation**. (*UML Distilled*, Ch. 5.)

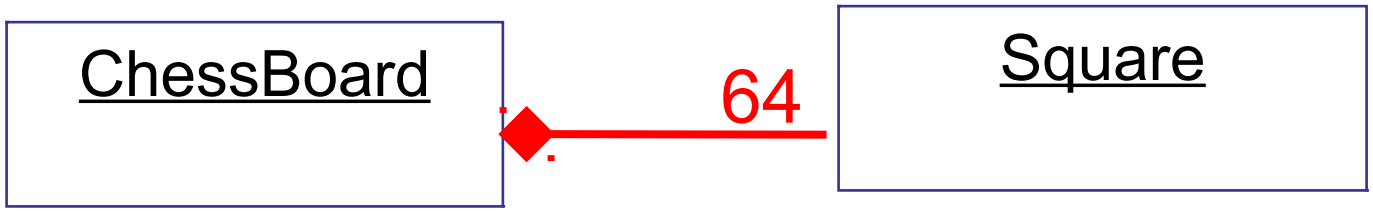
Just show it as **association**.

If it is really "composition" then show composition.

Composition: ownership relation

One class "owns" objects of the other class.

If the "whole" is destroyed, the parts are destroyed, too.




```
public class ChessBoard {
   private Square[][] squares = new Square[8][8];
```

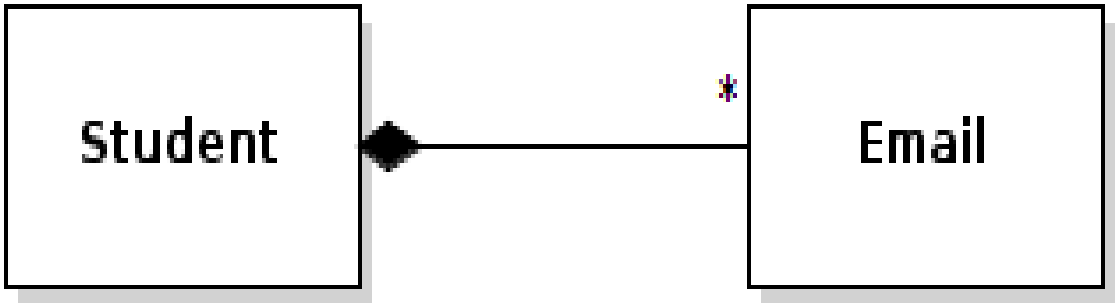
A Student owns his Email Addresses

Composition: A Student *owns* his Email addresses.

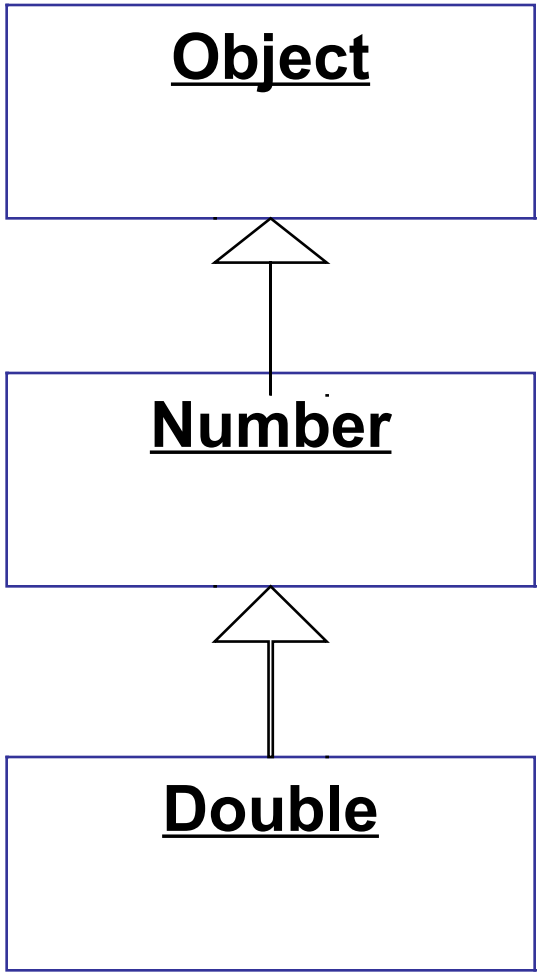
1) No one else can have the same email address.

2) When he is destroyed, we destroy his addresses, too!

```
public class Student {
    /** student uniquely owns his email addresses*/
    private List<Email> emailAddress;
```



Inheritance



Number is a *subclass* of Object.
Number *inherits* all the methods of Object. But, it *overrides* the definition of some methods, and adds new methods..

Double is a subclass of Number.
Double *inherits* all the methods of Number.
Double *overrides* the definition of some methods, and adds new methods.

Other names for Inheritance

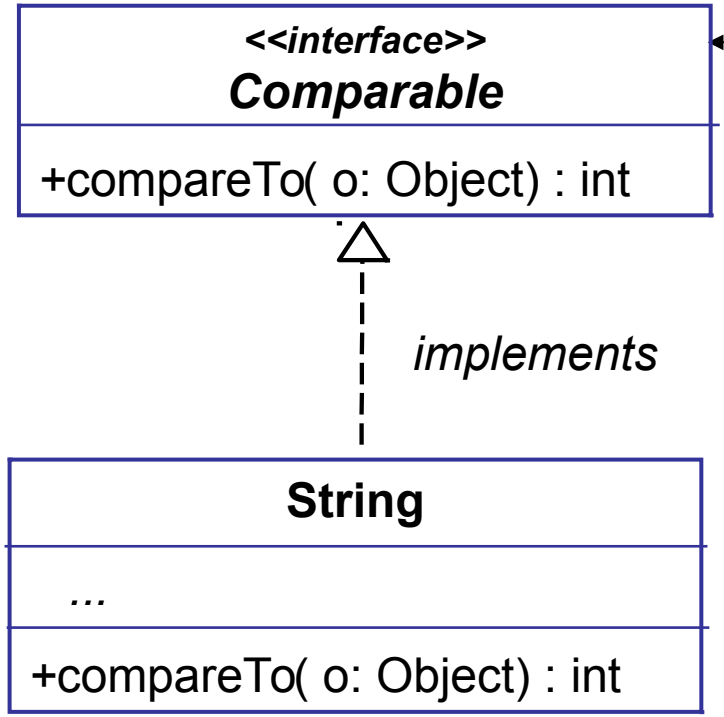
Specialization - a subclass is a *specialization* of the superclass.

Generalization - the superclass *generalizes* behavior of a hierarchy of subclasses.

Implements an Interface

Interface

The **`String`** class *implements* **`Comparable`** interface



write <<interface>> above the name

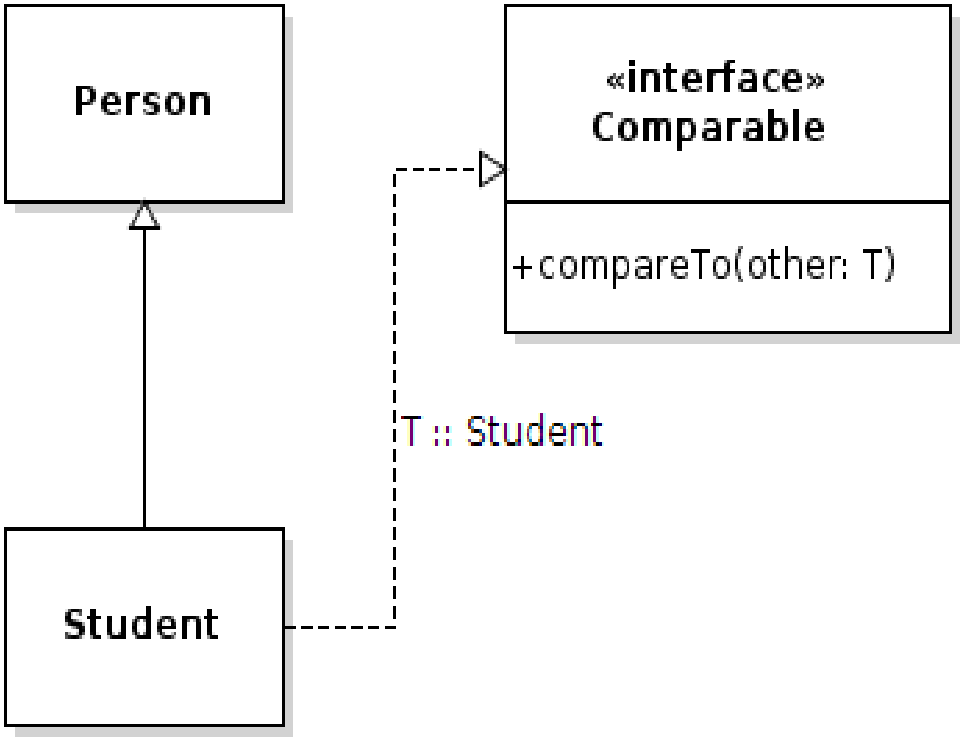
You don't need to write "implements" on the diagram,

but you MUST use a dashed arrow and triangle arrowhead as shown here.

Inheritance & Implements

You can have both in one class.

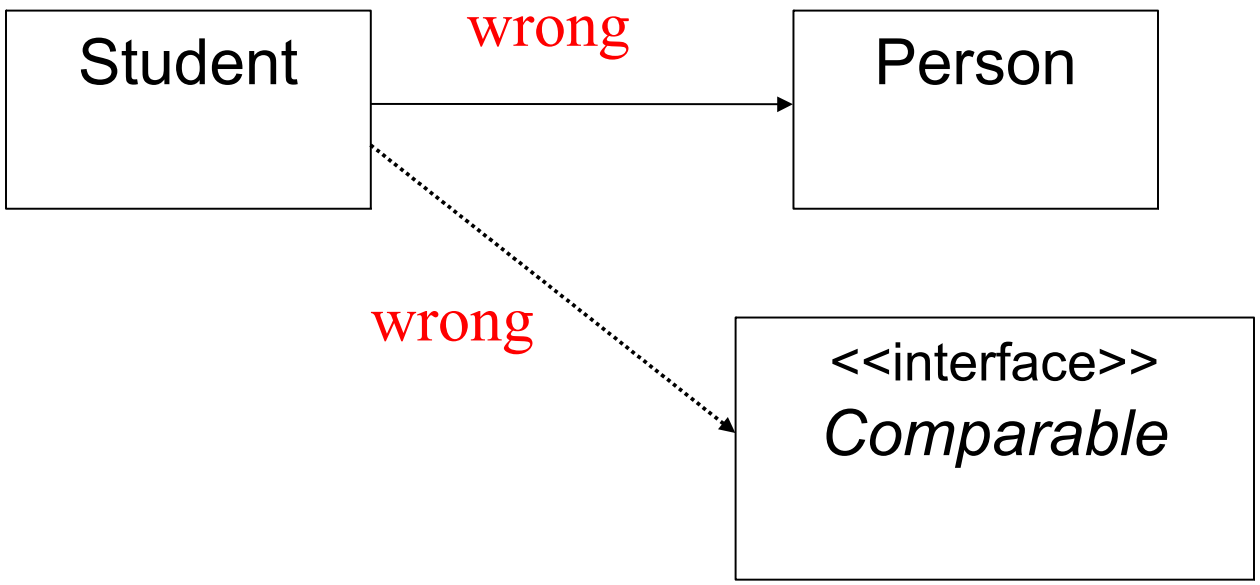
```
public class Student extends Person
         implements Comparable<Student> {
```



Errors

To communicate clearly, use the **correct notation**.

Example: in a circuit diagram, if you draw a diode in the wrong direction, the circuit might explode.



No partial credit for wrong relationships or incorrect notation.

Exercise: draw UML class diagram

A Coin has a value (double).

```
class Coin {
    private double value;
    public Coin(double value) ...;
    public double getValue( ) { ... }
    public boolean equals(Object other) { ... }
}
```

A Purse contains zero or more coins:

```
class Purse {
    private List<Coin> list = new ArrayList<Coin>();
    public boolean add(Coin coin) {...}
}
```

Exercise: Guessing Game

Draw UML class diagram of Guessing Game w/ relations

Include the classes:

`NumberGame` - superclass of all games

* see the lab sheet for attributes and methods

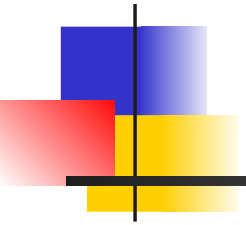
`GuessingGame` - subclass of `NumberGame`

`GameConsole` - plays a NumberGame

* has only one method: play(NumberGame game)

* which class does it depend on?

`Main` - creates the objects



Designing with UML

Extra features useful for showing design

Association Names

You can write text on the middle of an association to show the nature or meaning of the association.
Useful for design, but not required.

Roles

You can write the name of the association on the *opposite end of the association*. Useful for design.

- For a Person, *Company* is the "employer".
- For a Company, *Person* is an "employee".