CS 2113
Software Engineering

Do this now!!!

Lecture 6:
Java Objects and Classes

```
git clone https://github.com/cs2113f16/lec-6.git
```

Professor Tim Wood - The George Washington University

Last Time

- Moving from C to Java
  - Basic syntax, object creation
- Project 1:
  - Linked Lists
  - How's it going?
  - Start coding ASAP!
- Get help:
  - Piazza, email
  - Prof. Wood: **Tuesday 11am-1pm**
  - TA: **Monday 10:10-11am** and **1:30-2:10pm** TOMP405
  - CS Study Hall: **Sunday 1-3pm** in Tompkins 402
  - UTF hours: **Wed 9:30-11:30am**, **Friday 1-3pm** in SEH4040

10 hours per week!

This Time

- Java Objects and Classes
  - Objected oriented vs Procedural programming
  - Access control
  - Inheritance

- Basic UML Diagrams
  - Representing code visually
  - Class Diagrams

The Java docs

- The main documentation of the Java language is in the form of "javadocs"
  - Created from informative comments in the actual source code

- Java 1.5   AKA Java 5 (kind of outdated)
  - http://download.oracle.com/javase/1.5.0/docs/api/
- **Java 1.6   AKA Java 6 (very common)**
  - http://download.oracle.com/javase/6/docs/api/
- Java 1.7   AKA Java 7 (kind of buggy)
  - http://download.oracle.com/javase/7/docs/api/

Practice reading these!

Textbook Reading

- Read Chapters 1-4
  - Chap 1 is basic syntax and Chap 3 is variables
  - Most of this should be review, so you can skim
- Focus on chapters 2 and 4
  - Object oriented programming, pass by value, encapsulation
- Do this before the next lab!

Procedural vs Object Oriented

- Programming is about **functions** and **data**
  - In C (a *procedural* language) those were kept separate:

```
typedef struct {
  char* name;
  double salary;
} employee;

double getTaxRate(employee e) {...}
```

- Procedural programming focuses on **what needs to be done**
- Object oriented programming focuses on **the data** that must be manipulated

Nouns vs Verbs

- Procedural is about actions (verbs)
- O.O. is about things (**nouns**)
- A simple program:

Write a program that asks for a **number** and then prints its square root.

- A more realistic example:

A ***voice mail system*** records calls to multiple ***mailboxes***. The system records each ***message***, the ***caller's number***, and the ***time of the call***. The ***owner*** of a mailbox can play back saved messages.

Benefits of O.O.P.

- Focuses on **Data** and **Relationships**

- Break functionality of program down into interacting objects
  - Divide and conquer

- Clearly defined interfaces between components
  - Fewer bugs from unexpected interactions
  - Ask an object to modify itself

- Supports hierarchies of objects

Objects and Classes in Java

- In Java:
  - A class is type of object
  - All objects have a class
  - Primitives (int, float, etc) and functions are not objects

```
public class Hello {
   public static void main (){
         System.out.println("hi.");
      }
}
```

- Classes contain data and functions
  - An object instantiates that data

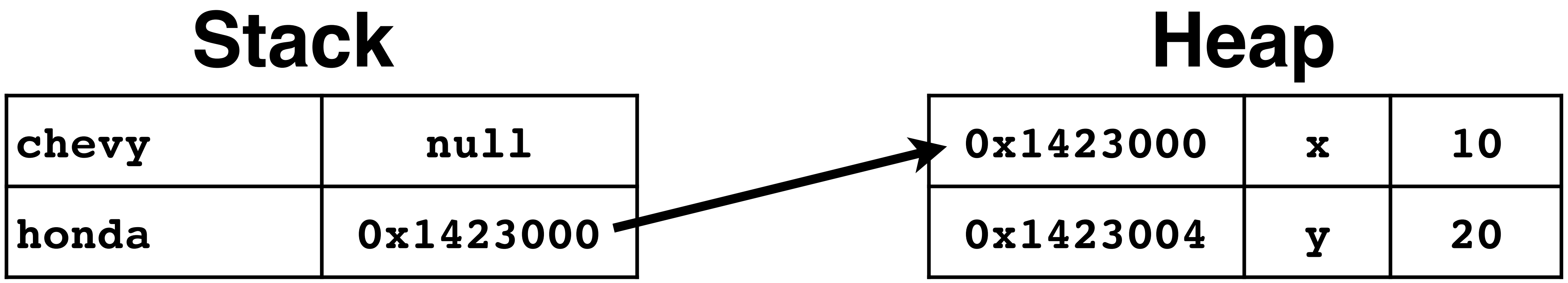
- Class Constructor
  - Used to create new objects
  - Initialize variables
  - Must call **`super(...)`** if a subclass

```
public class Car {
   public int x;
   public int y;

   public Car(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public void setX(int x) {
      this.x = x;
   }
```

Creating new Objects

- To use an object we need a reference to it
- Use the **`new`** command to instantiate an object
  - Reserves memory on the Heap for that object class
- Each new object gets its own chunk of memory
  - But they share functions and static class variables

```
public static void main(...)
{
   Car chevy;
   Car honda;
   honda = new Car(10, 20);
```



Stack

chevy	null
honda	0x1423000

Heap

0x1423000	x	10
0x1423004	y	20

Racing

- Get the code and run it (**press ctrl-c to quit)**:

```
git clone https://github.com/cs2113f16/lec-6.git
cd lec-6/proc
javac SimpleTrack.java
java SimpleTrack
```

- Simple changes
  - Green car
  - Different car symbol
  - Faster car
- Harder changes:
  - Have two cars instead of one
    - (Don't add any new classes yet)

`static` Members

- **`static`** Functions and Data are not specific to any one object instance
  - One copy is kept for the entire class
- You can access **`static`** members with just the class name
  - Do not need a reference to a specific object
- Useful for globally available data or functions that do not depend on any one object instance

Class name

Object name

```
Math.random() // static method in Math class
Math.PI // static data object in Math class

Car myCar = new Car(10,10);
myCar.x // instance variable
```

Uses of **static**

- #1: Great for classes that cannot be instantiated:
  - `Math` class
    - Provides general utility methods, doesn't have any data that must be unique for each object instance. You never do "new Math()"
- #2: Useful for sharing a single variable for all instances of a class
  - `carsBuilt` counter
    - Isn't unique to each car
    - Shouldn't need a reference to a car object to find out how many have been made
- Just don't abuse it

```java
public class Car
{
  public static int carsBuilt;
  public String license;
  public Car(){
    carsBuilt++;
    // ....
  }
}
```

```java
Car c = new Car();
System.out.println(c.license);
System.out.println(Car.carsBuilt);
```

Modularizing the code

- **Car** class: a "V" drawn to the screen that randomly moves back and forth. Starts at random position.
  - Data?
  - Functions?

Work in lec-6/oop

- **RaceTrack** class: instantiates a car object and has the main run loop of the program

```
Loop forever...
  update the car's position
  for each horizontal position in the line...
      if the car is at that position, draw it
      else draw an empty space
  print newline character and reset the color
  sleep for 30 milliseconds
```

Lots of objects

- How do we make an array of objects?

```
Car cars[10];

for(int i=0; i < cars.length; i++) {
  cars[i].update();
}
```

Lots of objects

- How do we make an array of objects?

```
Car cars[10];

for(int i=0; i < cars.length; i++) {
  cars[i] = new Car();
}

for(Car c: cars) { // fancy array looping
  c.update();
}
```

Lots of cars!

- Modify your program so it can create 3 cars at random positions and show them driving around

- Too easy? Try these:
  - Each car can be a different color
  - Make the left and right sides of the race track dynamically change width and don't let any cars go beyond them
  - If two cars collide, they should explode (i.e., display X and stop printing those two cars from then on)
  - Make the program stop when only one car is left

Inheritance

- Classes can be defined in a hierarchy
- The child (subclass) inherits data and functionality from its parent (superclass)
- Use the **`extends`** keyword in java

```java
public class RocketCar extends Car {
  public int numEngines; // define new class members

  public RocketCar(int x, int y, int n) {
    super(x,y); // must call parent's constructor
    numEngines = n;
  }
  public void draw()
  {
    drawSmokeTrail();
    super.draw(); // optionally call parent version
  }
}
```

Functions

- Inherited from parent class
  - Can be replaced fully, or can call **`super.funcname()`** to combine parent and child functionality
- Supports multiple declarations with different parameters

```
public void draw(Color c)
{
  // draw with specific color
}
public void draw()
{
  // this code is unnecessary
  super.draw()
}
```

Vehicle Types

- Extend your program to define multiple types of vehicles or objects with inheritance

- Some options:
  - Wild cars move more erratically than regular race cars
  - Some cars look different from others
  - Trees only appear on some lines
  - Walls are at the edges of the track and move less erratically than cars. If a car hits a wall it crashes
  - ...something more creative that you think of...

Time check: ~4:45

Limiting Visibility

- Goal of OOP: compartmentalize data and function

- Classes can be:
  - **`public`** - visible everywhere
  - (no keyword) - visible within package

- Data and functions can be:
  - **`public`** - callable, readable/writable by anyone
  - **`private`** - only accessible within the same class
  - **`protected`** - accessible in the class, subclass, and package
  - (no keyword) - accessible within the package

Visibility

- Answer question 1 on the worksheet.

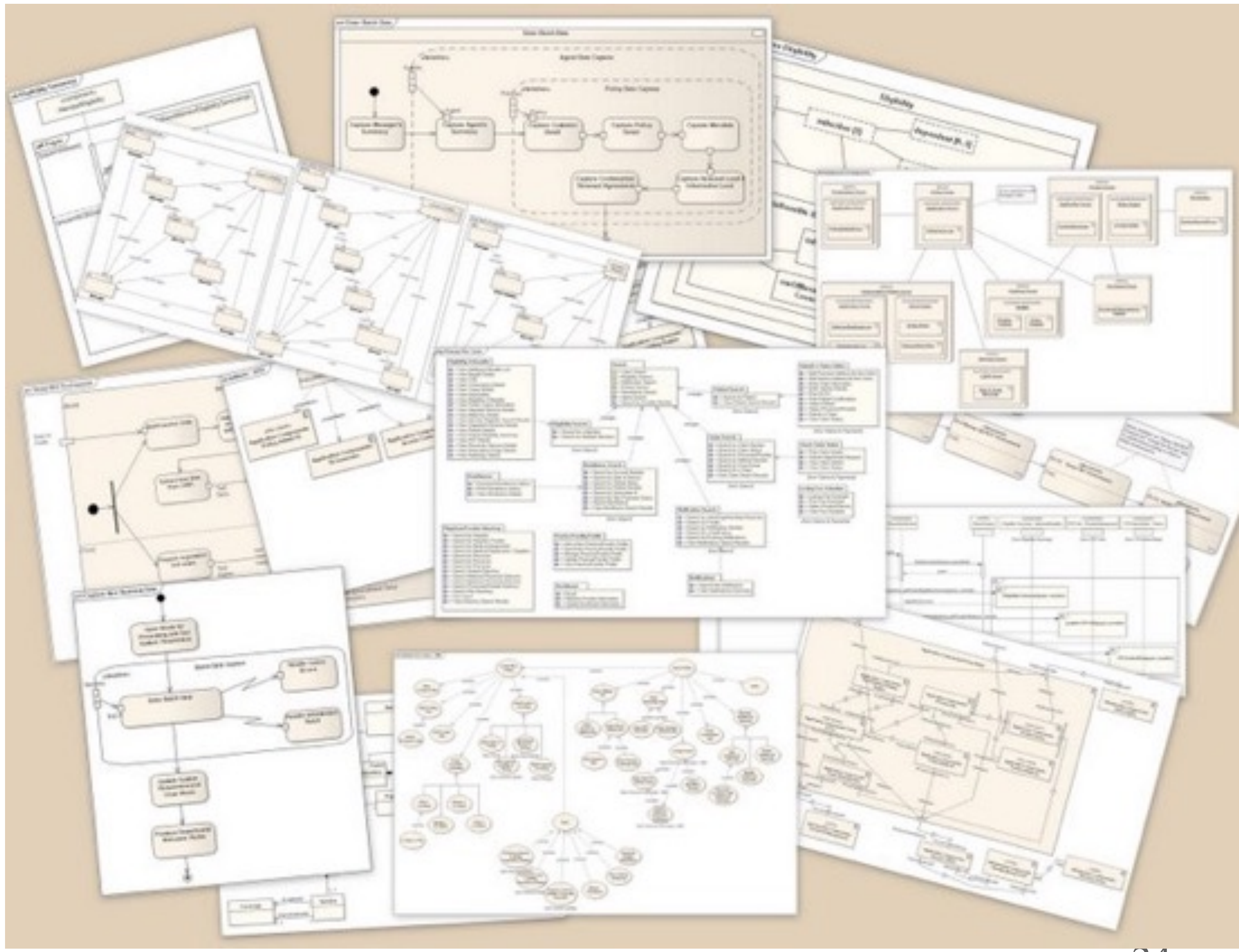
- Data and functions can be:
  - **`public`** - callable, readable/writable by anyone
  - **`private`** - only accessible within the same class
  - **`protected`** - accessible in the class, subclass, and package
  - (no keyword) - accessible within the package

O.O. Design

- Modern software engineering is largely about designing classes, deciding how they relate, and deciding their functions + data
- Good design:
  - **is simple**: remove unnecessary classes, functions, and data
  - **is compartmentalized**: separate functionality, isolate data
  - **has clean interfaces**: inputs and outputs should make sense
  - **is reusable**: create general purpose code when possible

UML

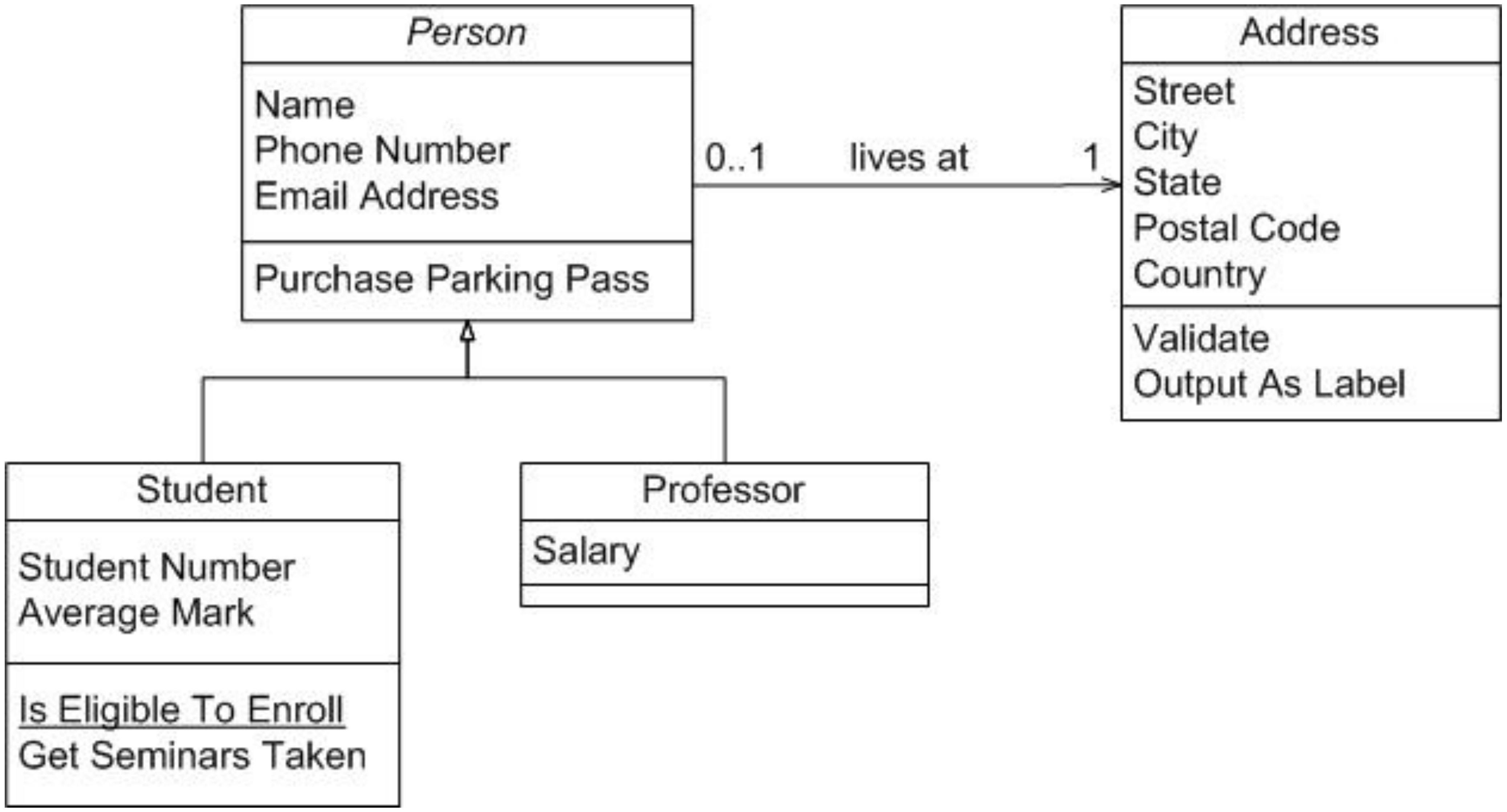
- Unified Modeling Language
  - Formal way to describe programs, components, functionality
  - Designed for any object oriented programming language
- Defines 14 standard diagram types



- Structural diagrams
  - Defines the pieces of the system and their relationships
- Behavioral diagrams
  - Describe how the pieces interact

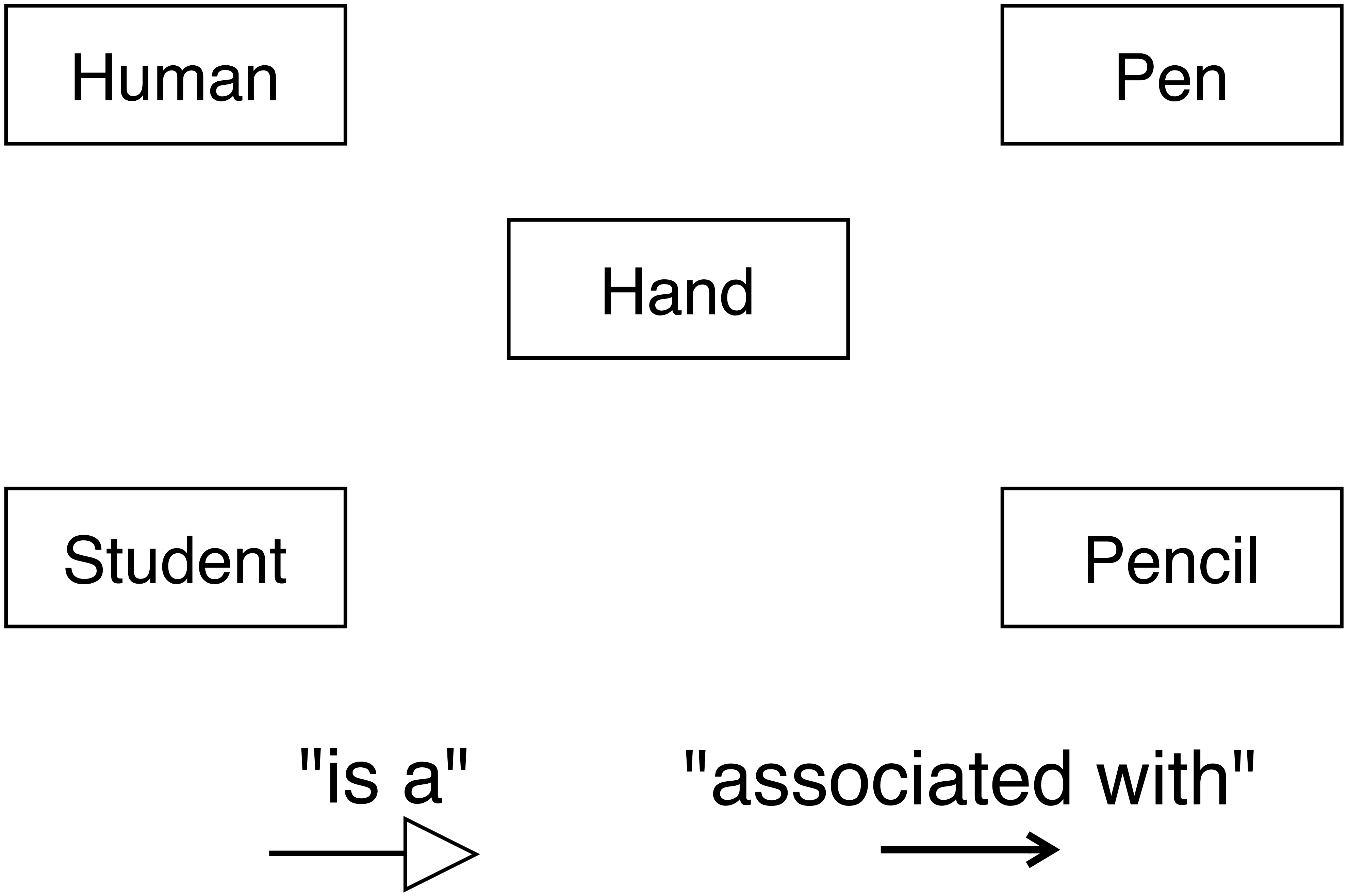
Class Diagrams

- Tell us how different classes are related
- Lists key methods and data



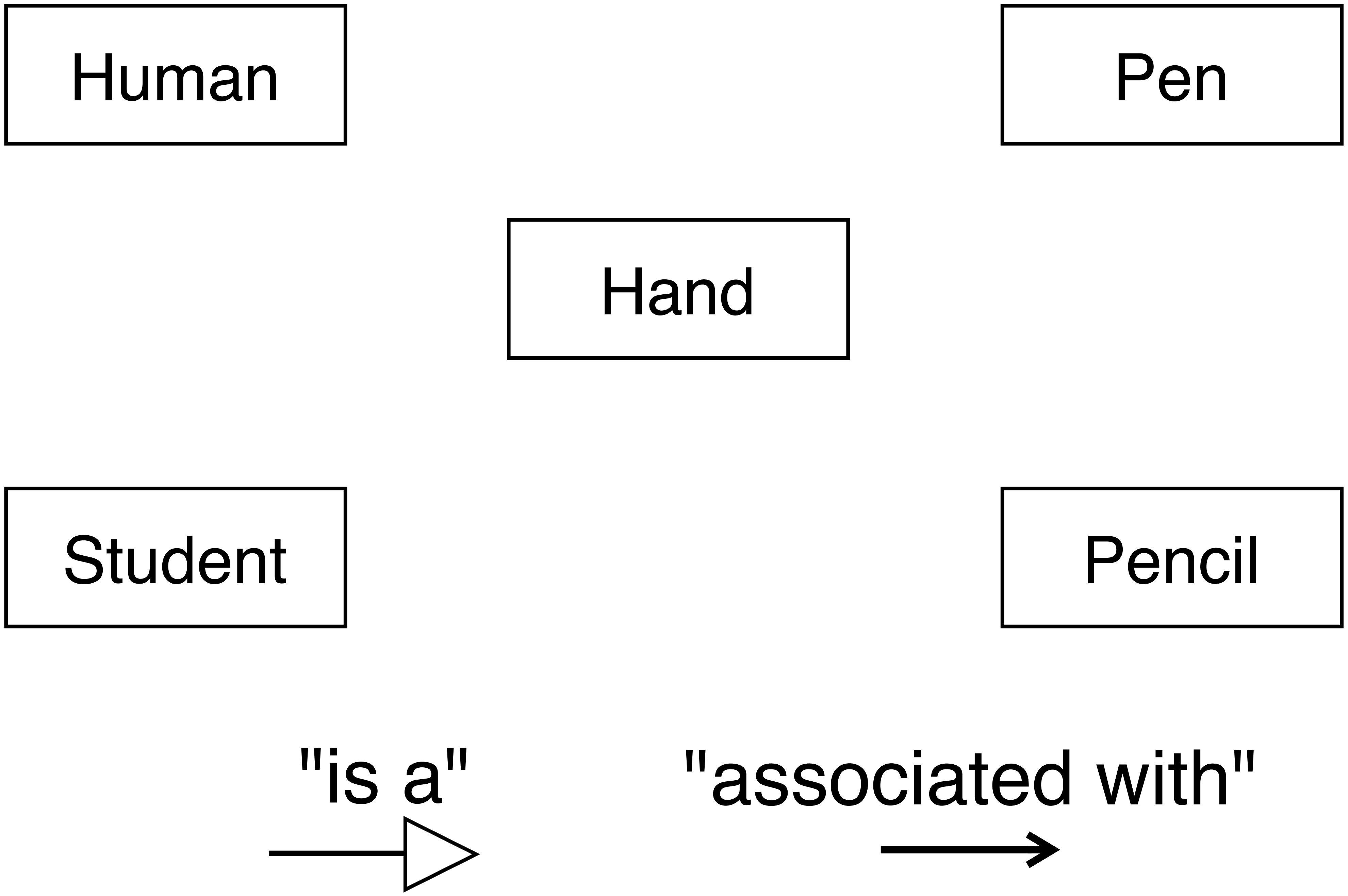
Simplified Class Diagrams

- How are these related?
  - Human, Student, Hand, Pen, and Pencil?



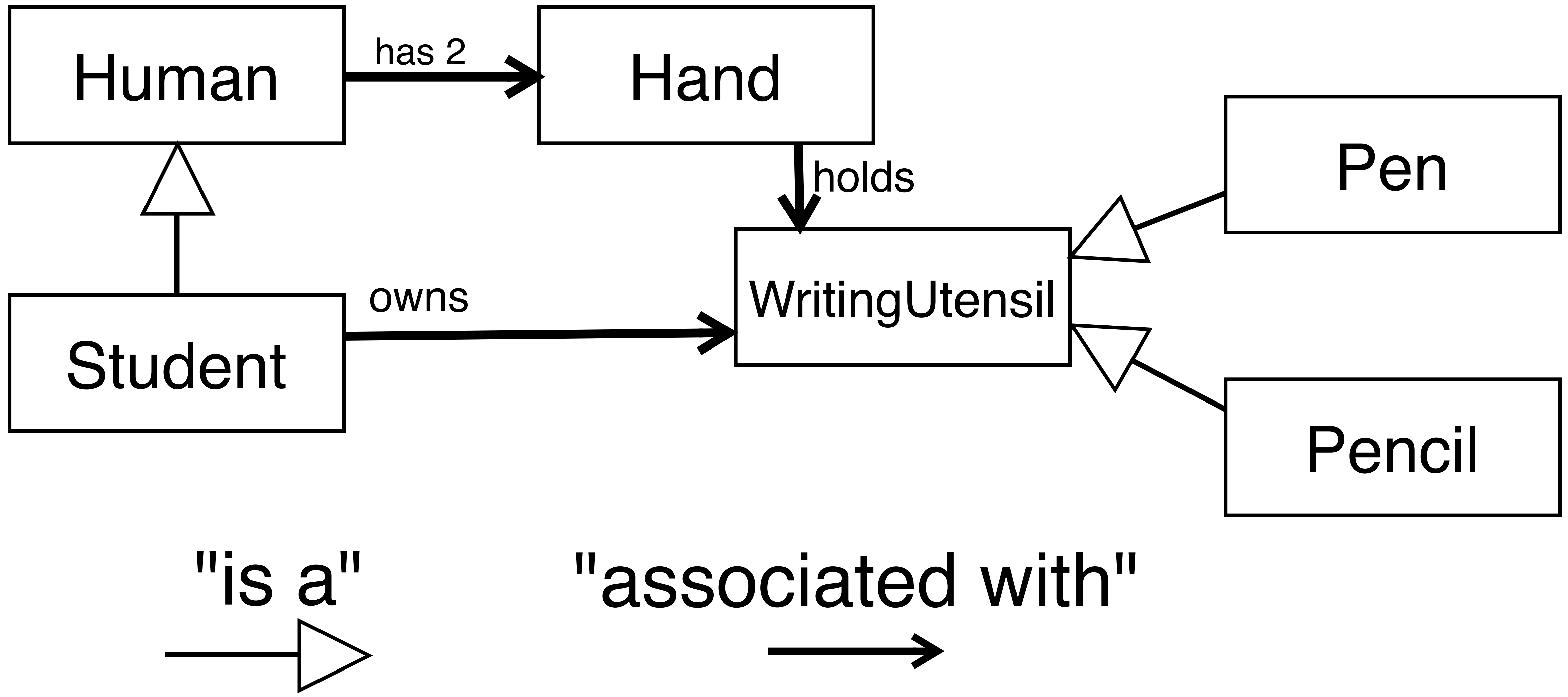
Simplified Class Diagrams

- How are these related?
  - Human, Student, Hand, Pen, and Pencil?



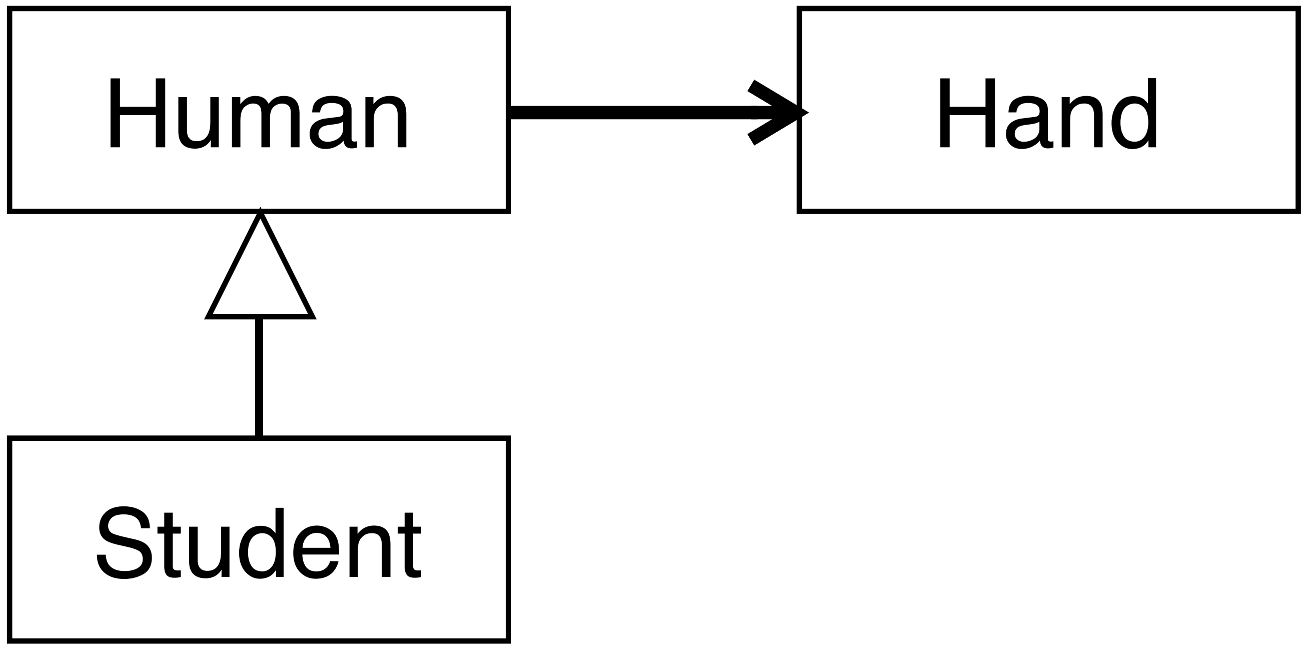
Class Diagram Example

Class diagrams describe the software's components and how they relate

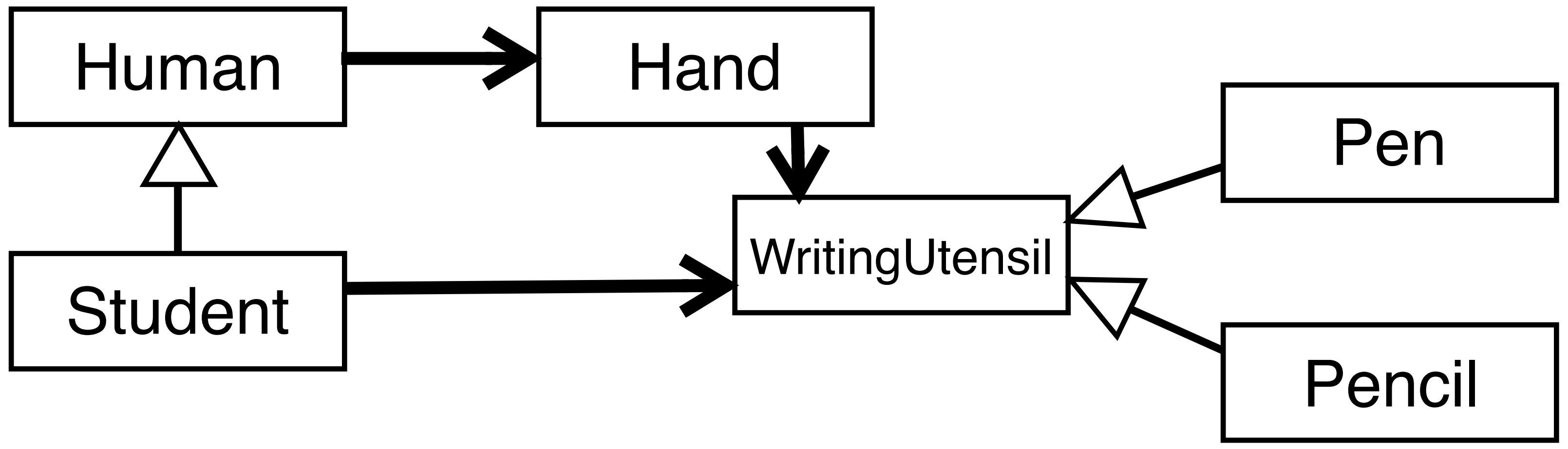


Arrow Direction

- The arrow shows which class must know something about the other
- A child must "know" about its parent so it can extend it
  - The parent can be oblivious to that fact



From Diagram to Code

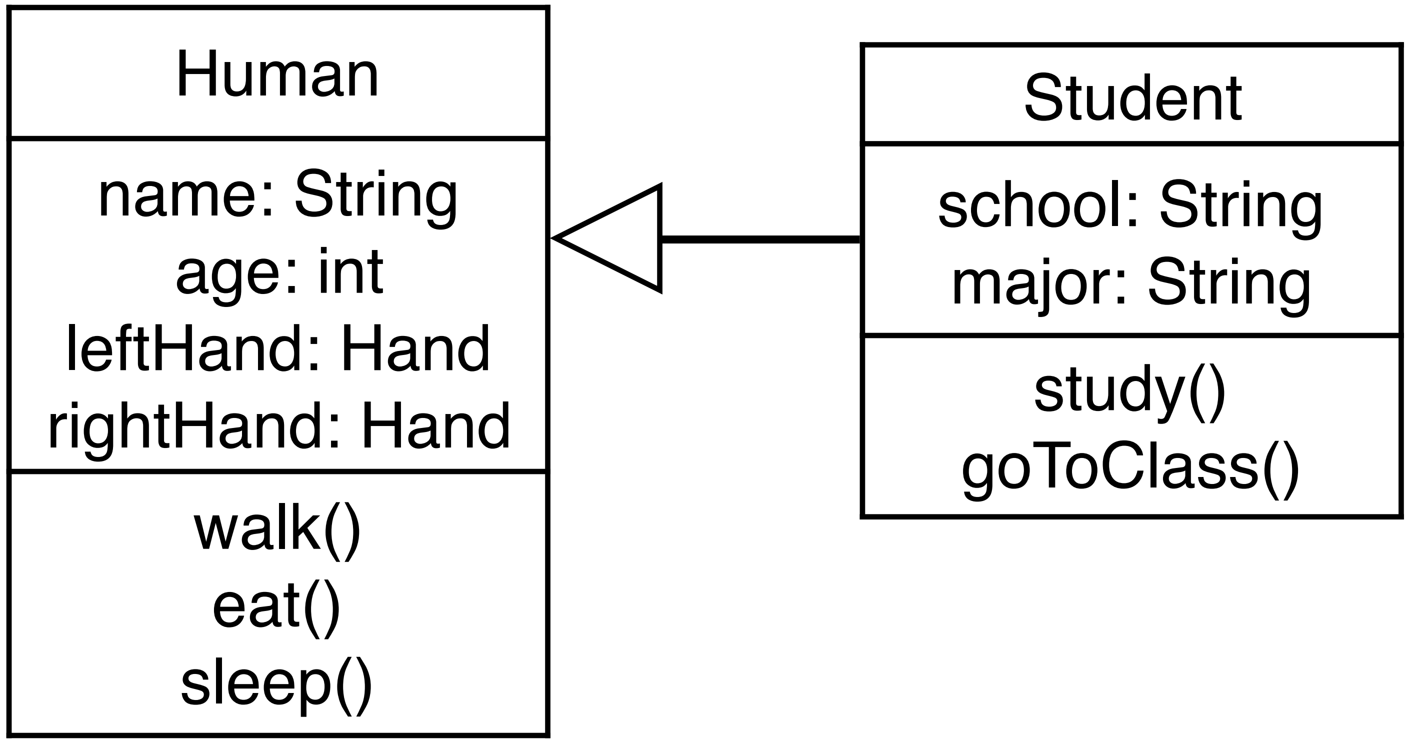


```
public class Human {
  protected Hand leftHand;
  protected Hand rightHand;

  public Human() {
    leftHand = new Hand();
    rightHand = new Hand();
  }
}
```

```
public class Student
    extends Human
{
  private WritingUtensil wu;

  public Student() {
    super();
    wu = new Pen();
    rightHand = new Hand(wu);
  }
}
```

Adding More Detail

- Full UML diagrams also include:
  - Attributes (class data elements)
  - Methods (class functions)



lists the data and functions **added** to the parent class

- Also uses a whole bunch of arrow types

Voice Mail System

- Problem description:

> Phone rings, is not picked up, caller invited to leave message; owner of mailbox can later retrieve message

- Is that enough to build the system?
- Requirements Engineering
  - Science behind formally specifying what a system must do

Deciding on the parts

- Things:
  - Mailboxes
  - Messages
  - Users
  - Phone numbers
  - Dates

A **voice mail system** records calls to multiple **mailboxes.** The system records each **message,** the **caller's number,** and the **time of the call.** The **owner** of a mailbox can play back saved messages.

- Actions:
  - Record message
  - Replay message
  - ...
  - Remove messages?
  - Next/Previous Message?

MailSystem Class Diagram

- What are the components?
  - What are their important functions and data?
- How are they related?
  - Annotate arrows

Work with 1-2 other students and draw your diagram on the worksheet

Inherits from

—▷

Makes use of

→

MailSystem Class Diagram

- What are the components?
  - What are their important functions and data?
- How are they related?
  - Annotate arrows

Client request: We also want to support text messages on the same system!

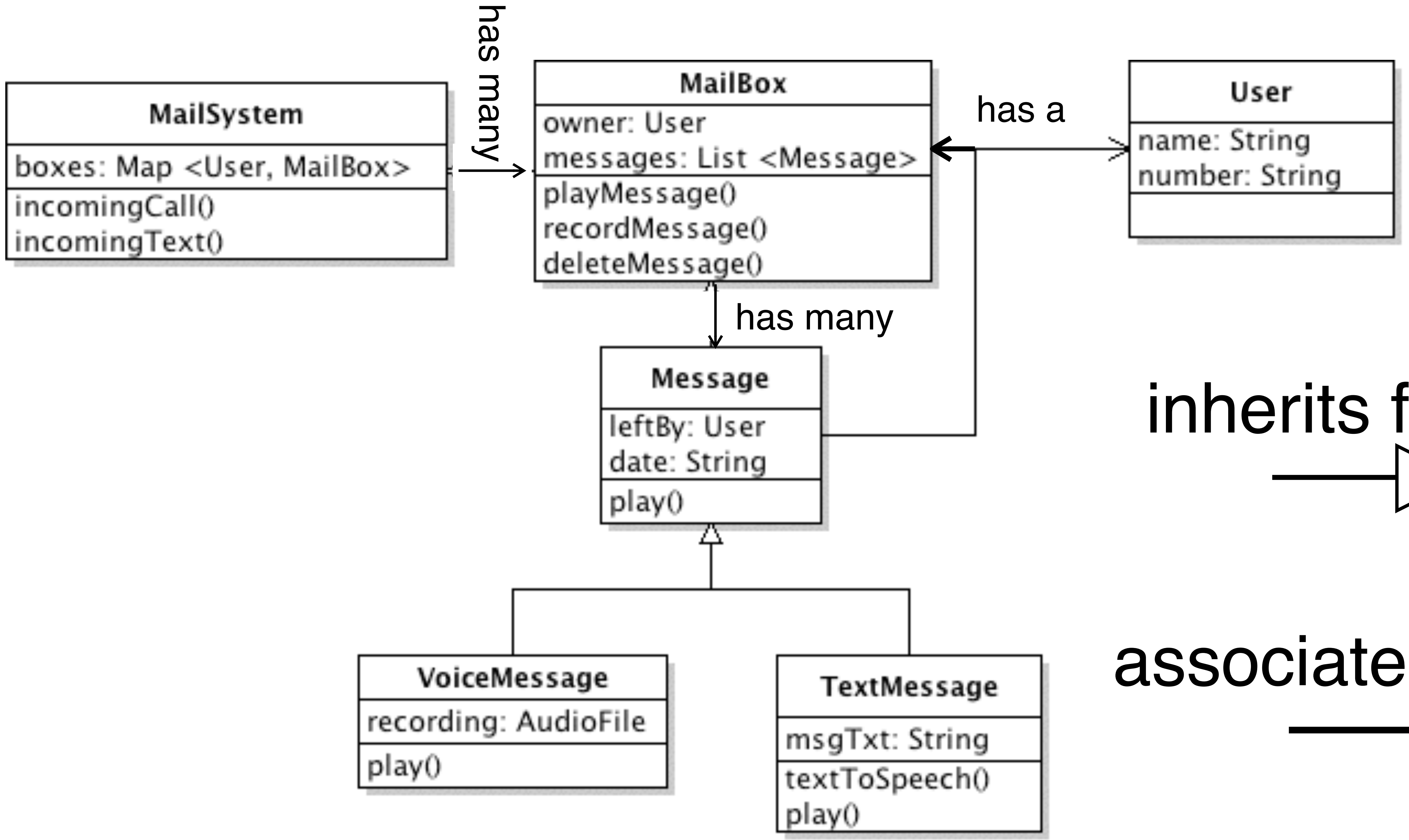
Inherits from

—▷

Makes use of

$\longrightarrow$

MailSystem Class Diagram



inherits from

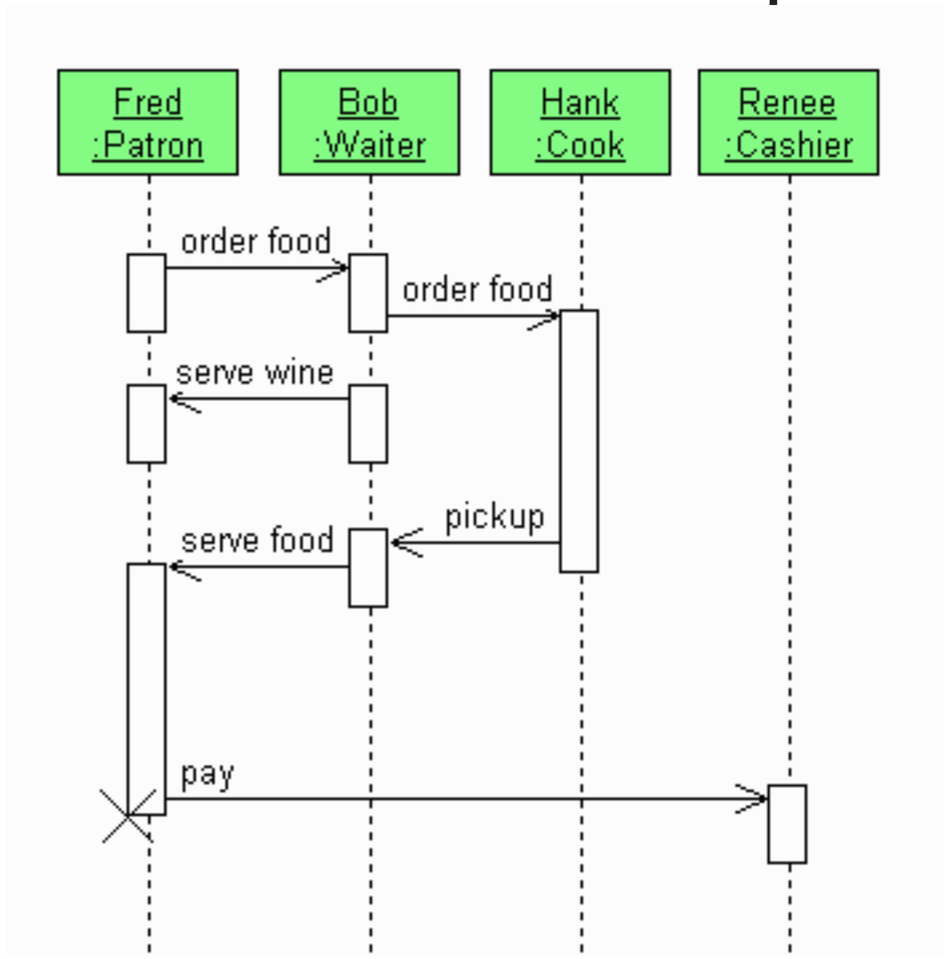
$\longrightarrow\!\triangleright$

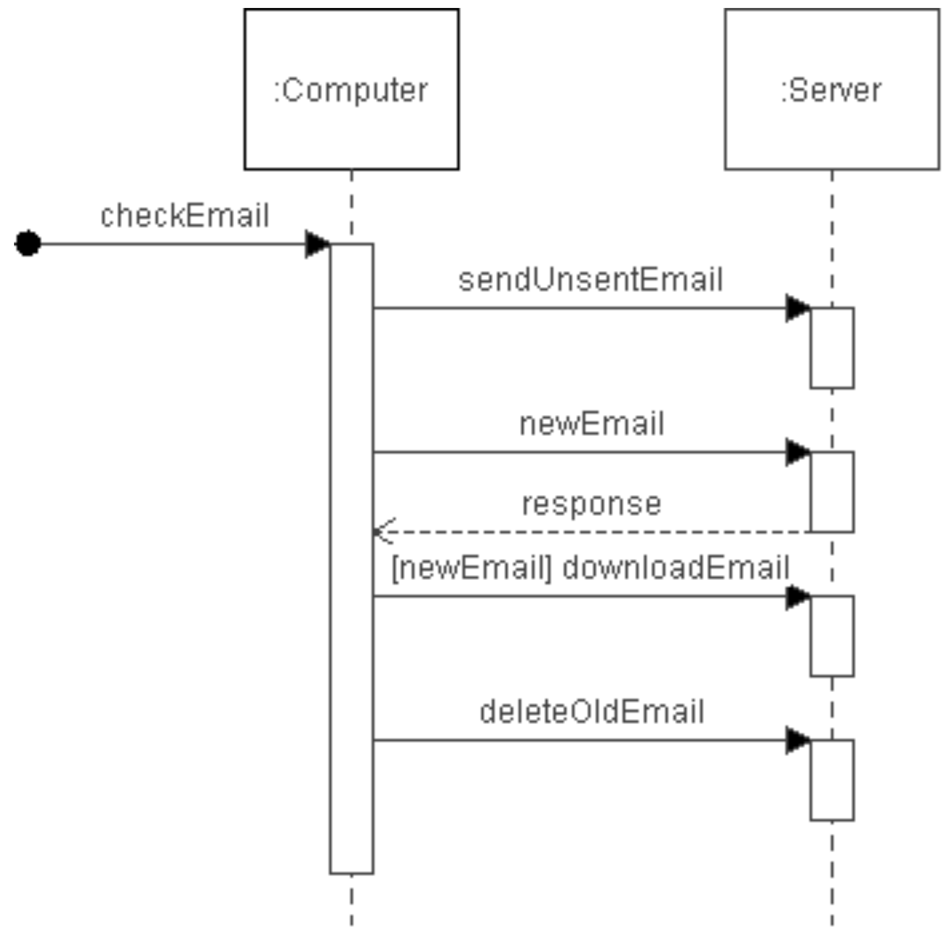
associated with

$\longrightarrow$

How do things interact?

- Class diagram tells *who interacts with who*
  - But doesn't illustrate *how they interact*
- UML Sequence Diagrams
  - Show the steps required to do something



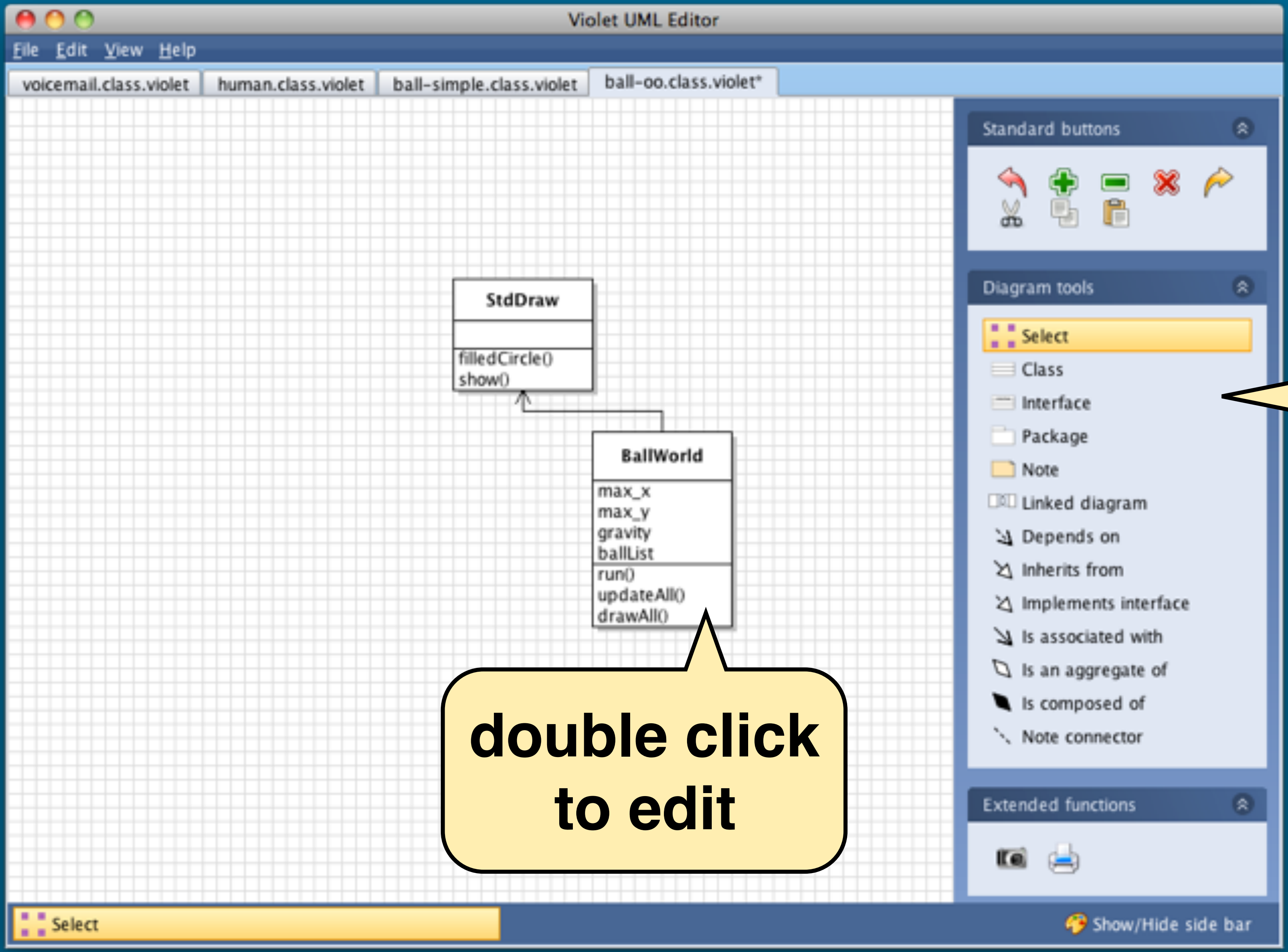


Good Practices

- Diagram should be loosely connected
  - Only make a class depend on another if it really needs it
- Use Inheritance to replace similar functionality
  - Red balls, green balls, and orange squares
  - Chickens, cows, and tanks
- Focus on key features and level of detail
  - UML can be a waste of time, so apply to the useful parts
- **Iterate design and implementation**
  - Don't try to do everything the first time
  - Build and test components in stages

UML Tool

- Violet UML tool makes it easy to draw diagrams
  - Homepage: http://alexdp.free.fr/violetumleditor/
  - **Download from 2113 course website!**



double click to edit

choose mode:
Select
Class
Inherits from...
Is associated...

Banking

- Use VioletUML to represent the following program

You have been hired by a bank to write software to track its accounts and clients. Your bank software must keep track of two kinds of accounts: Checking and Savings. Both these account types should support making deposits and withdraws. The Checking account should also allow customers to write a check and the Savings account should support adding interest.

The bank needs to be able to keep track of all of its customers, each of whom may have one or more accounts. Every month, the bank needs to be able to print out a list of customers and the balance inside each of their accounts. You can assume that all customers have a unique name and that bank accounts are assigned a unique ID number.

Draw a UML diagram to represent the software you would design to handle this scenario. Be sure to mark the important functions and data members for each class, and use the different arrow types to indicate which classes inherit or are associated with others.

Summary

- Java emphasizes Object Oriented Programming
- We use OOP to write better structured code
  - A correct program is always better than a fast buggy one
- Think about how you organize classes
  - Use private variables and expose clean interfaces
  - Use inheritance to reuse code
- UML can help plan your code