Java 3: More OOP

CS 2113 Software Engineering

The George Washington University

Quiz time!

- Hopefully you finished Java Module 2 and read through the book!
- Solve question 1 on the worksheet

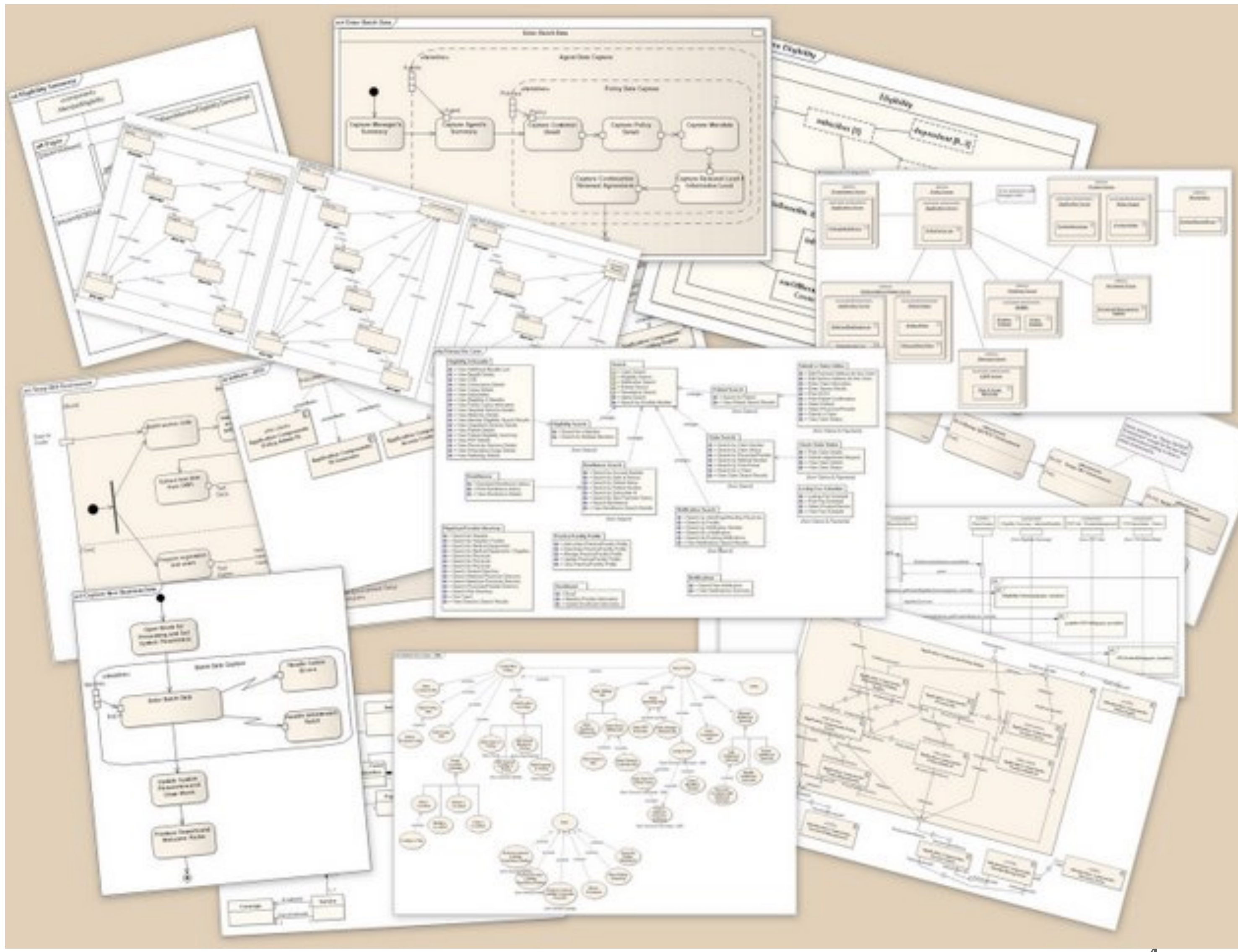
O.O. Design

- 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

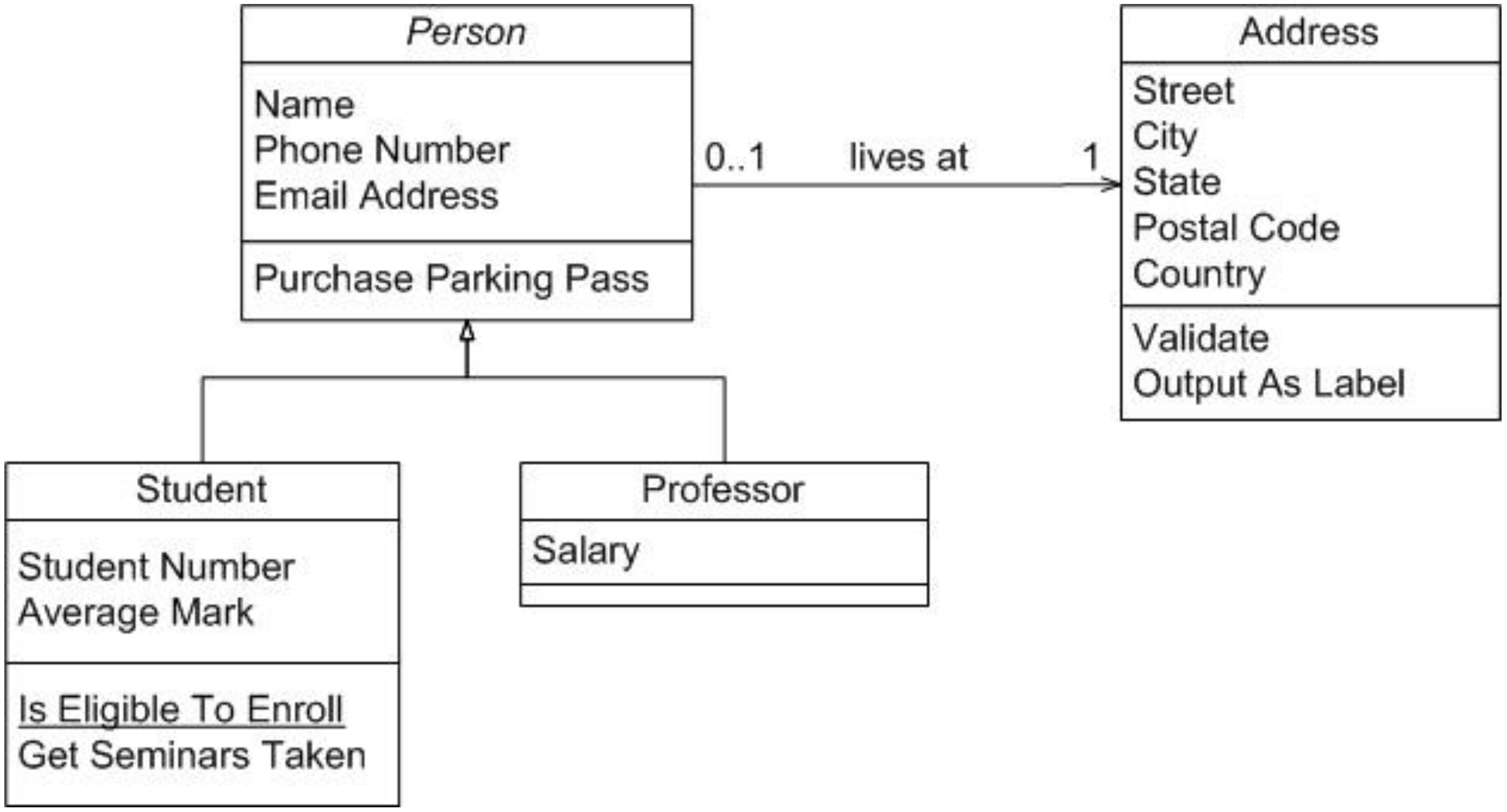
- Unified Modeling Language
  - Formal way to describe programs, components, functionality
  - Designed for any object oriented programming language
- Defines 14 standard diagram types

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



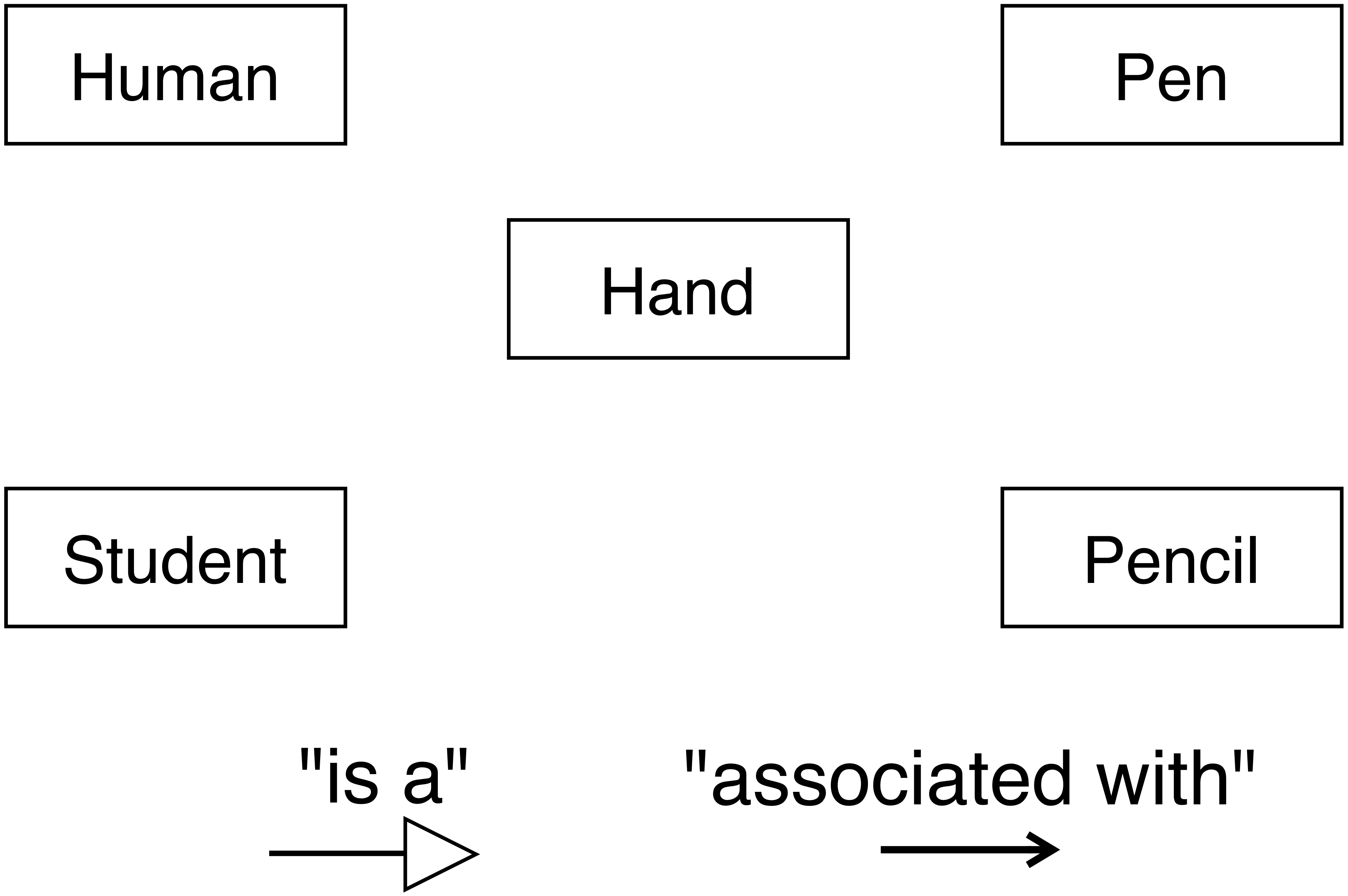
Class Diagrams

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



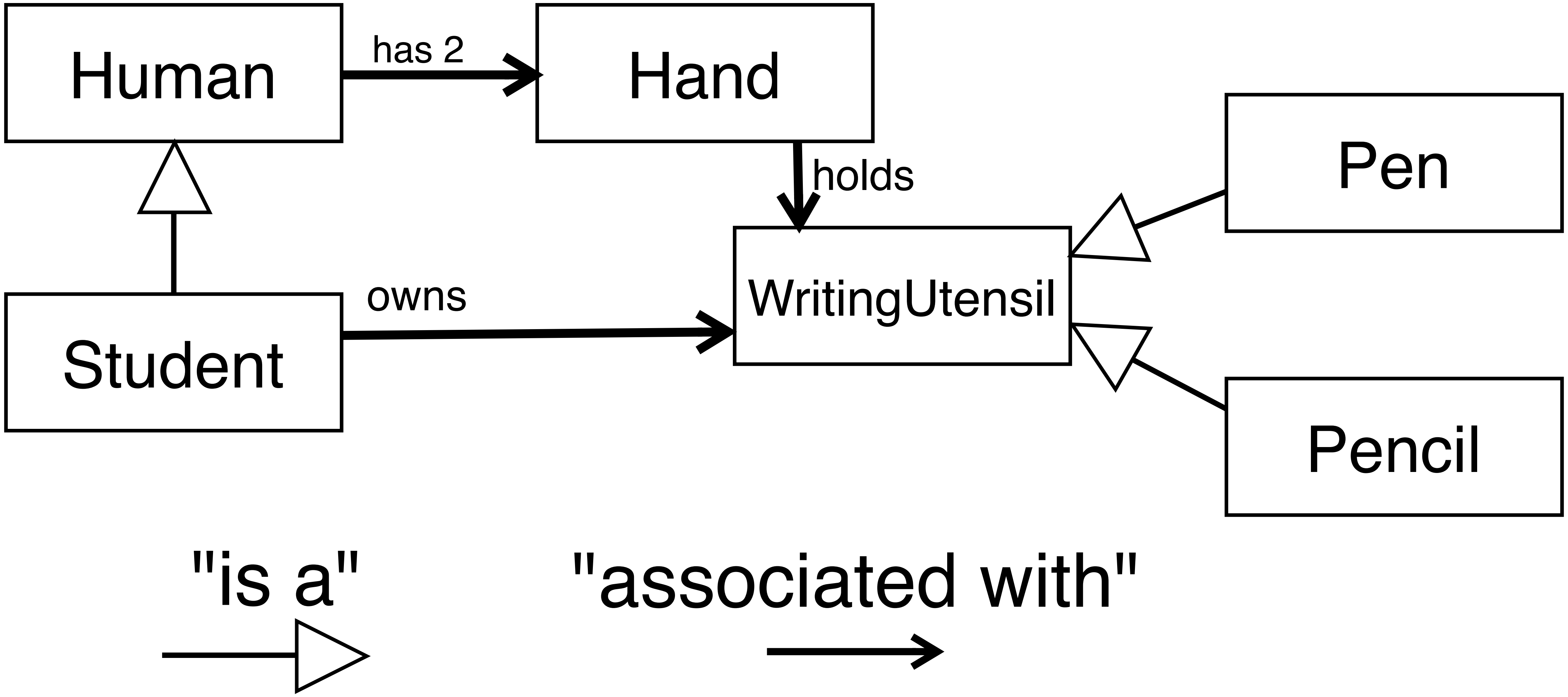
Simplified Class Diagrams

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



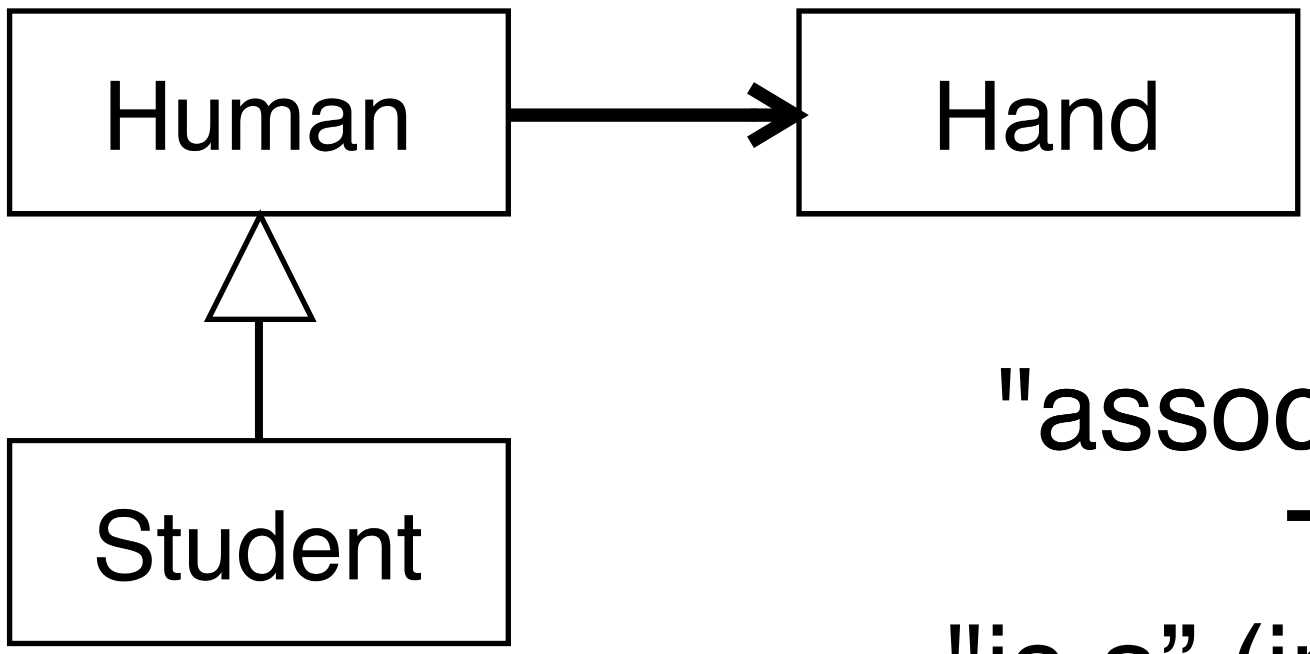
Class Diagram Example

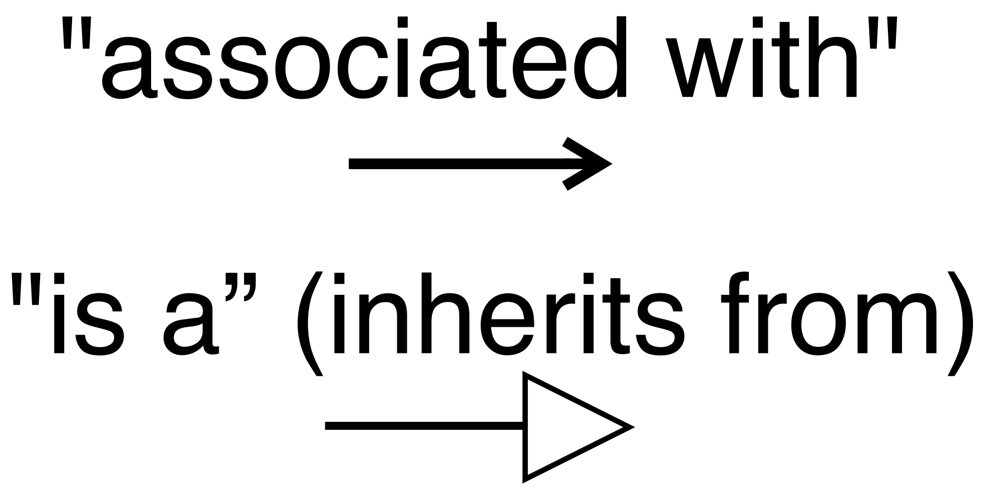
Class diagrams describe the software's components and how they relate



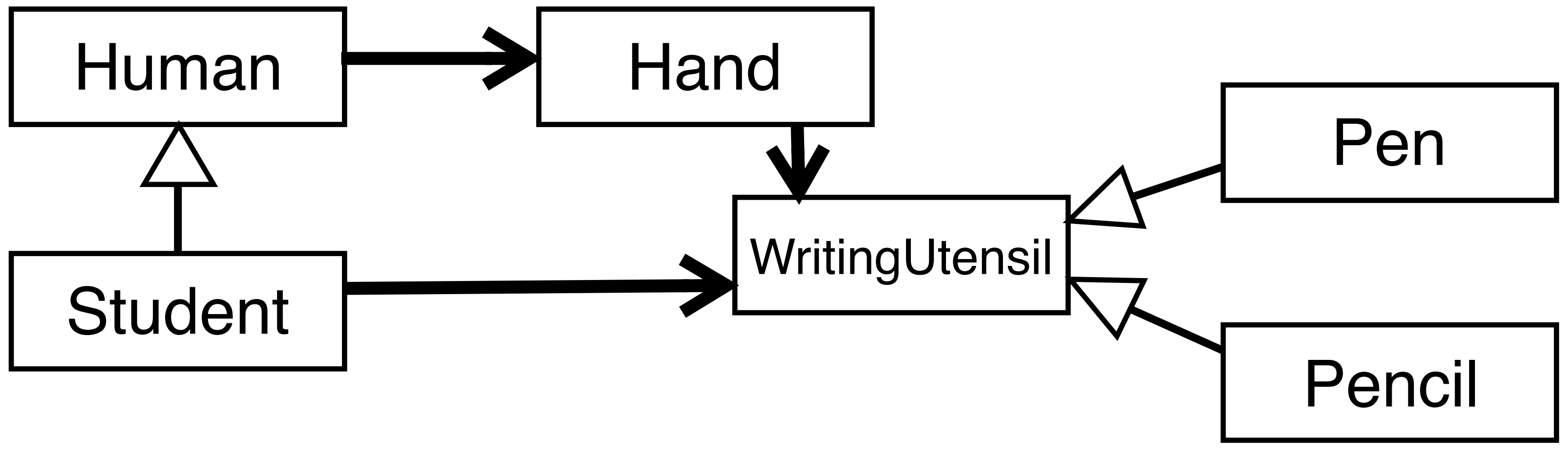
Arrow Direction

- 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

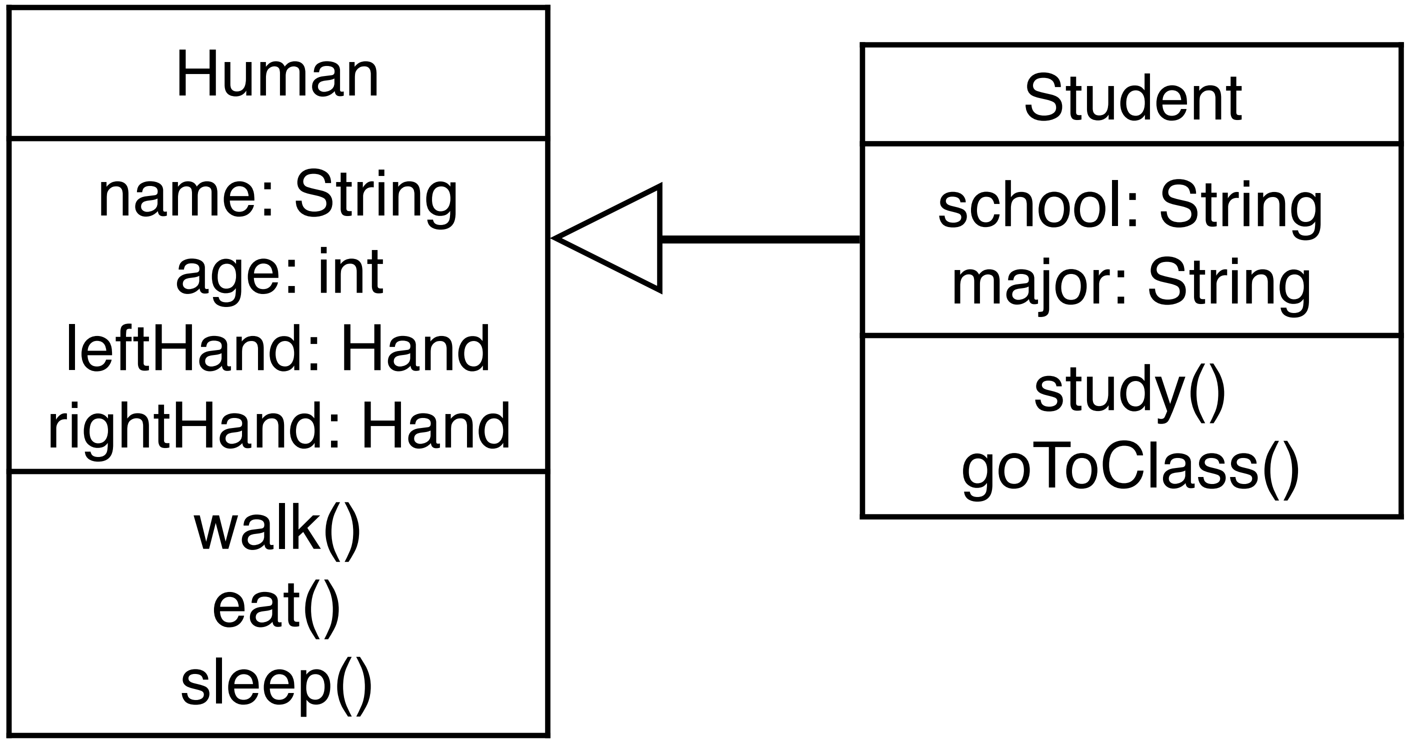


```
public class Human {
  protected Hand leftHand;
  protected Hand rightHand;

  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

$\longrightarrow$

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

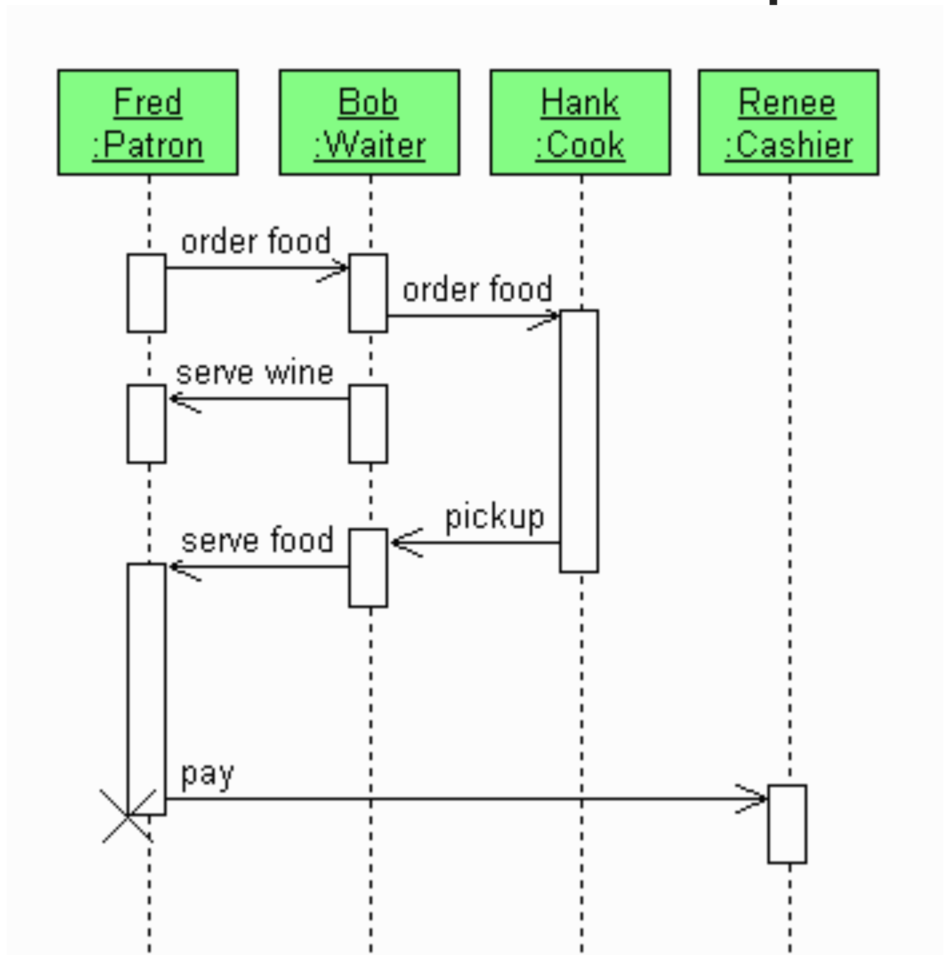
──▷

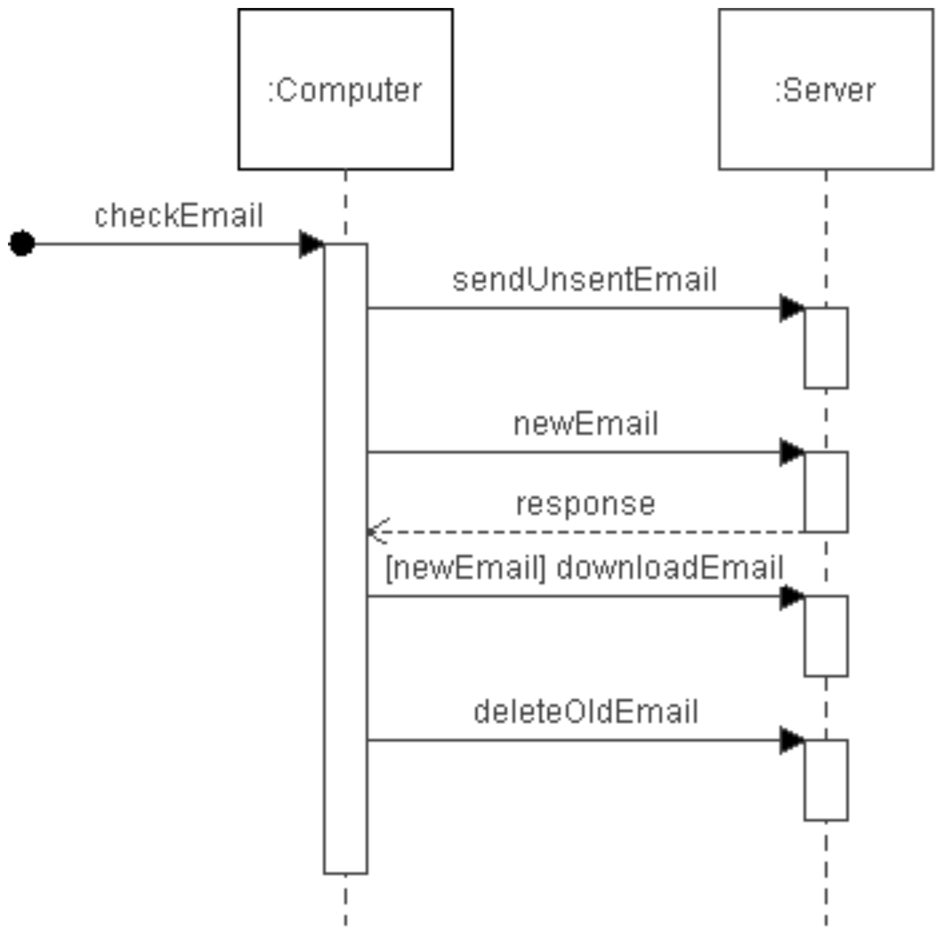
Makes use of

⟶

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



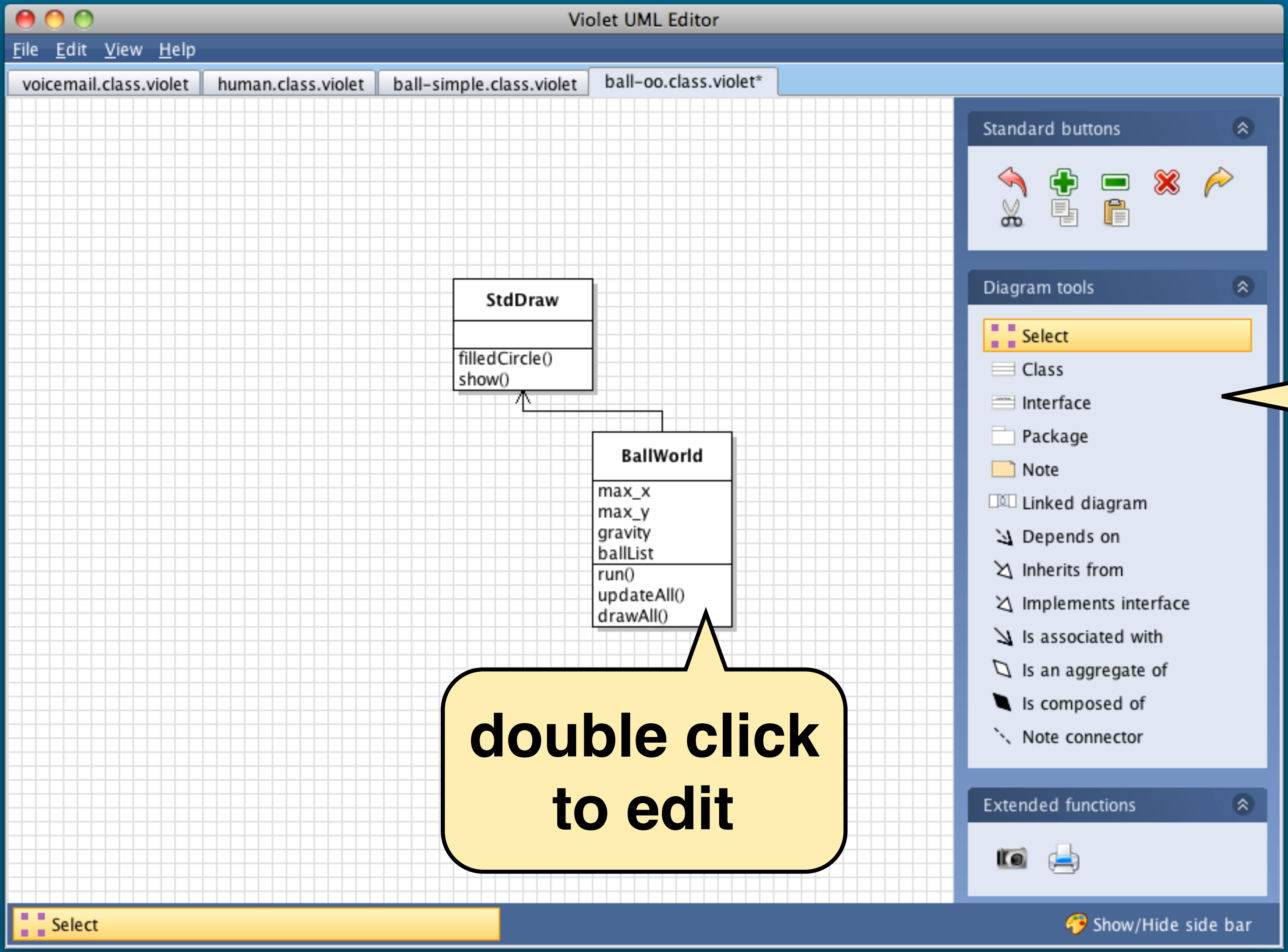


Good Practices

- 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 Java Module 3 page!**



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

Banking

- Use VioletUML to represent the following program

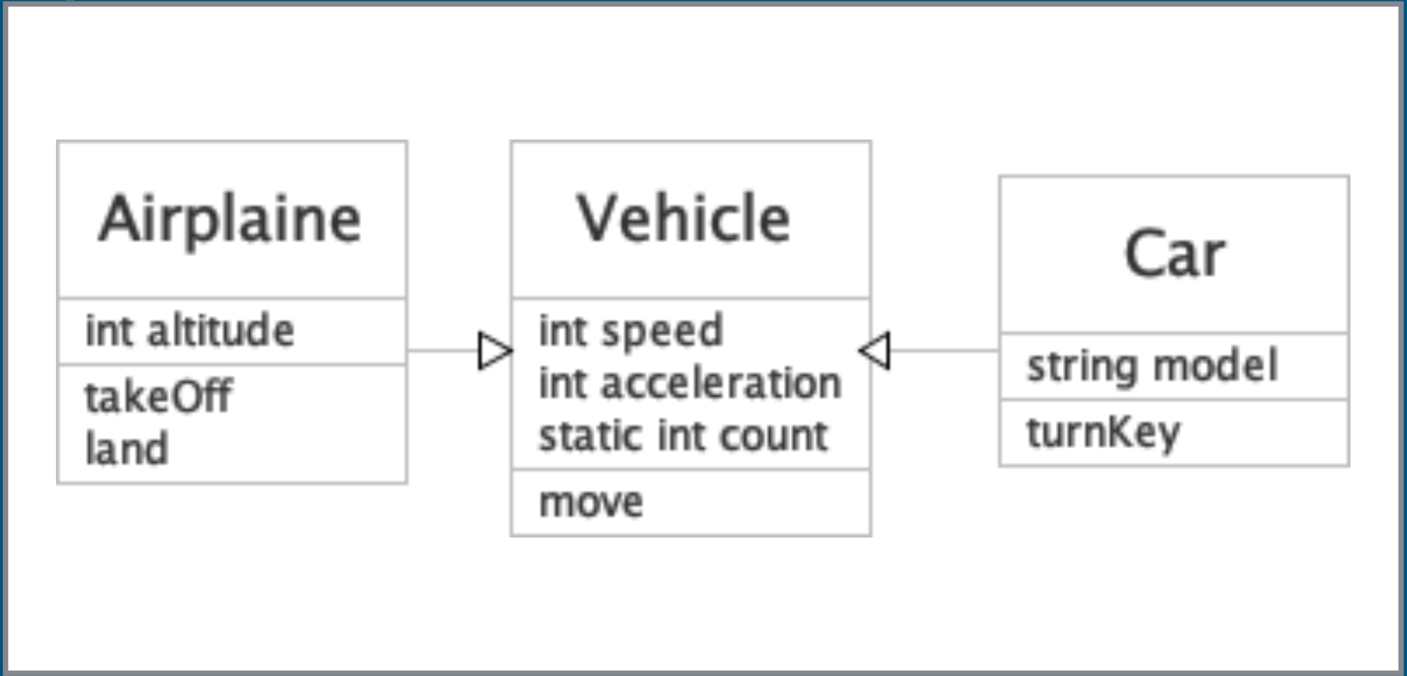
You have been hired by a bank to write software to track its accounts and clients. Your bank software must keep track of two kinds of accounts: Checking and Savings. Both these account types should support making deposits and withdraws. The Checking account should also allow customers to write a check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

Classes and Memory

- UML Class Diagrams help us visualize relations



- Write the code for these classes
  - Just define the functions and data members
- Fill in worksheet
  - Ignore the main function and memory layout for now…
  - You do not need to fill in the method content, just put the names of methods and variables in the right places

Classes, Objects, and Memory

- What will memory look like after running this code?

```
public class Vehicle {
   public int speed;
   private int acceleration;
   public static int count;
   // ...
}

public class Car extends Vehicle {
  private String model;
  public void turnKey(){ … }

  public static void main() {
    int s = 65;
    Vehicle v;
    Vehicle v2 = new Vehicle();
    v2.speed = s;
    Car c = new Car("Honda");
    c.count = 50;
  }
```

Stack		
Address	*Name*	*Contents*
10000	s	65
10008	v	
10016	v2	
10016	c	

Heap	
Address	*Contents*
500000	

Static Object Memory

Stack

Heap

Globals

```
// A simple static object. Note: all members (data and methods)
// are declared  with the "static" keyword.

class ObjX {

    static int i;

    static void print ()
    {
     System.out.println ("i=" + i);
    }

} // All static members

// The class that has "main"

public class StaticExample {

    public static void main (String[] argv)
    {
       // Refer to a member of ObjX using the class name.
     ObjX.i = 5;

      // Call a method using the class name and dot-operator.
     ObjX.print();
    }
}
```

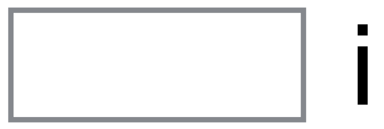
Static Object Memory

```
// A simple static object. Note: all members (data and methods)
// are declared  with the "static" keyword.

class ObjX {

    static int i;

    static void print ()
    {
     System.out.println ("i=" + i);
    }

} // All static members

// The class that has "main"

public class StaticExample {

    public static void main (String[] argv)
    {
       // Refer to a member of ObjX using the class name.
     ObjX.i = 5;

      // Call a method using the class name and dot-operator.
     ObjX.print();
    }
}
```

Stack

Heap

Globals



Dynamic Object Memory

Stack

Heap

Globals

```
// Dynamic object definition
class ObjX {

    int i;

    void print () {
      System.out.println ("i=" + i);
    }

} // NO “static” for either member!

public class DynamicExample1 {

    public static void main (String[] argv)
    {
      // First create an instance with space from the heap.
     ObjX x = new ObjX ();

      // Now access members via the variable and the dot-operator.
     x.i = 5;
     x.print();
    }
}
```

Static/Dynamic (HW)

Stack

```
// Static reference to dynamic object
class ObjX {

    int i;

    void print () {
      System.out.println ("i=" + i);
    }

} // same as before

public class DynamicExample2 {

    // A simple variable declaration.
    static ObjX x;

    public static void main (String[] argv)
    {
      // First create an instance, with space from the heap.
      x = new ObjX ();
      // Now access members via the variable and the dot-operator.
      x.i = 5;
      x.print();
    }
}
```

Heap

Globals

Multiple Objects

Stack

Heap

Globals

```
// Multiple dynamic objects
class ObjX {

    int i;

    void print () {
      System.out.println ("i=" + i);
    }

} // same as before

public class DynamicExample3 {

    public static void main (String[] argv)
    {
       // Create an instance and do stuff with it.
       ObjX x = new ObjX ();
       x.i = 5;
       x.print();

       // Create another instance assigned to the same variable.
       x = new ObjX ();
       x.i = 6;
       x.print();
    }

}
```

Multiple Objects 2 (HW)

Stack

Heap

Globals

```
// Multiple dynamic objects
class ObjX {

    int i;

    void print () {
      System.out.println ("i=" + i);
    }

} // same as before

public class DynamicExample4 {

    public static void main (String[] argv)
    {
      // Create an instance and do stuff with it.
      ObjX x = new ObjX ();
      x.i = 5;
      x.print();

      // Create another instance assigned to the same variable.
      ObjX x2 = new ObjX ();
      x2.i = 6;
      x2.print();
    }

}
```

Arrays of Objects (HW)

```
// Multiple dynamic objects
class ObjX {

// same as before
}

public class DynamicExample5 {
    public static void main (String[] argv)
    {
      // Make space for 4 ObjX pointers.
      ObjX[] xArray = new ObjX [4];
      // Make each of the 4 pointers point to ObjX instances.
      for (int k=0; k < 4; k++) {
            xArray[k] = new ObjX ();
      }
      // Now assign data to some of them.
      xArray[0].i = 5;
      xArray[1].i = 6;
      // Print all.
      for (ObjX x: xArray) {
            x.print();
      }
    }
}
```

Stack

Heap

Globals

Arrays of Objects v2 (HW)

Stack

```
// Multiple dynamic objects
class ObjX {

// same as before
}

public class DynamicExample6 {
    public static void main (String[] argv)
    {
      // Make space for 4 ObjX pointers.
      ObjX[] xArray = new ObjX [4];
      // Use fancy for loop syntax.
      for (ObjX x: xArray) {
            x = new ObjX ();
      }
      // Now assign data to some of them.
      xArray[0].i = 5;
      xArray[1].i = 6;
      // Print all using fancy for loop syntax.
      for (ObjX x: xArray) {
            x.print();
      }
    }
}
```

Heap

Globals

Where Does it Go?

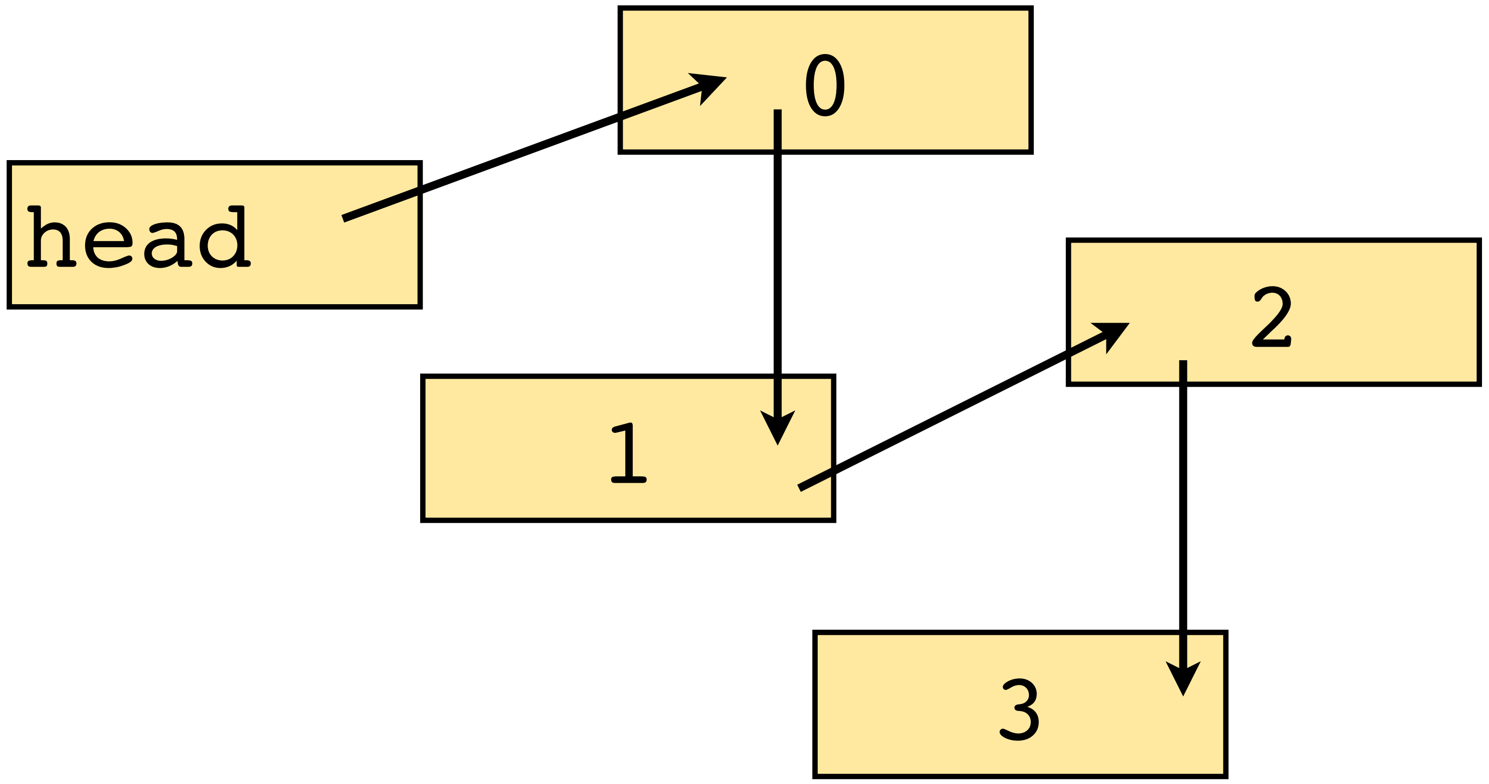
- In C we had to call "free()" to make sure that the memory we used was cleaned up
- How come we don't need to do this in Java?

```
MyLinkedList L =
  new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```

Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory

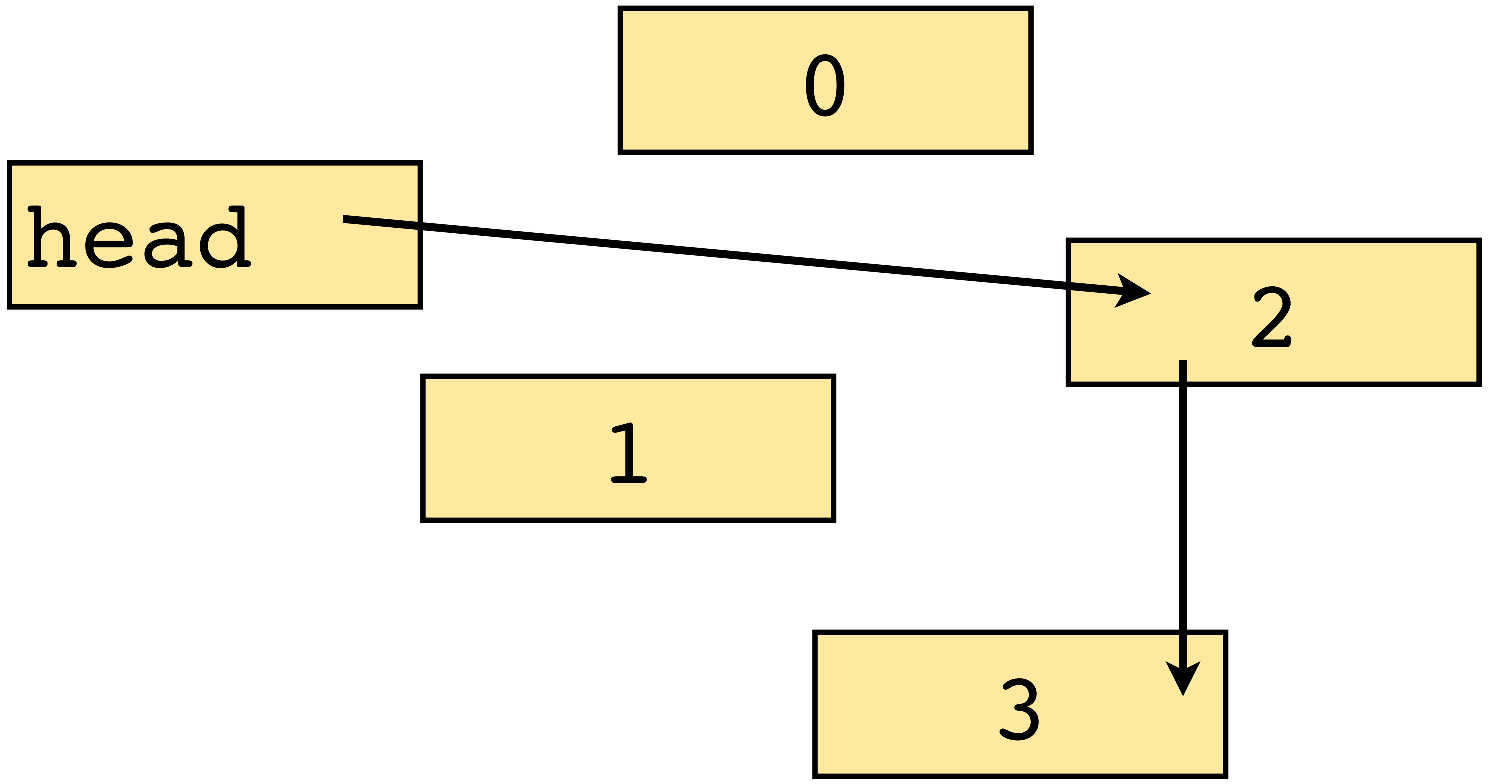
```
MyLinkedList L =
    new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```



Garbage Collection

- The Java Run Time automatically tracks what objects are actively being used in memory
  - If no variable references something, then that object is "lost"---can be deleted

```
MyLinkedList L =
  new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(0);
L.remItemAt(0);
```



How might it do this?

- How to find which objects on the heap are reachable (or not)?

- Program knows:

Garbage Collection

- The Java Run Time automatically tracks what objects are still used in memory

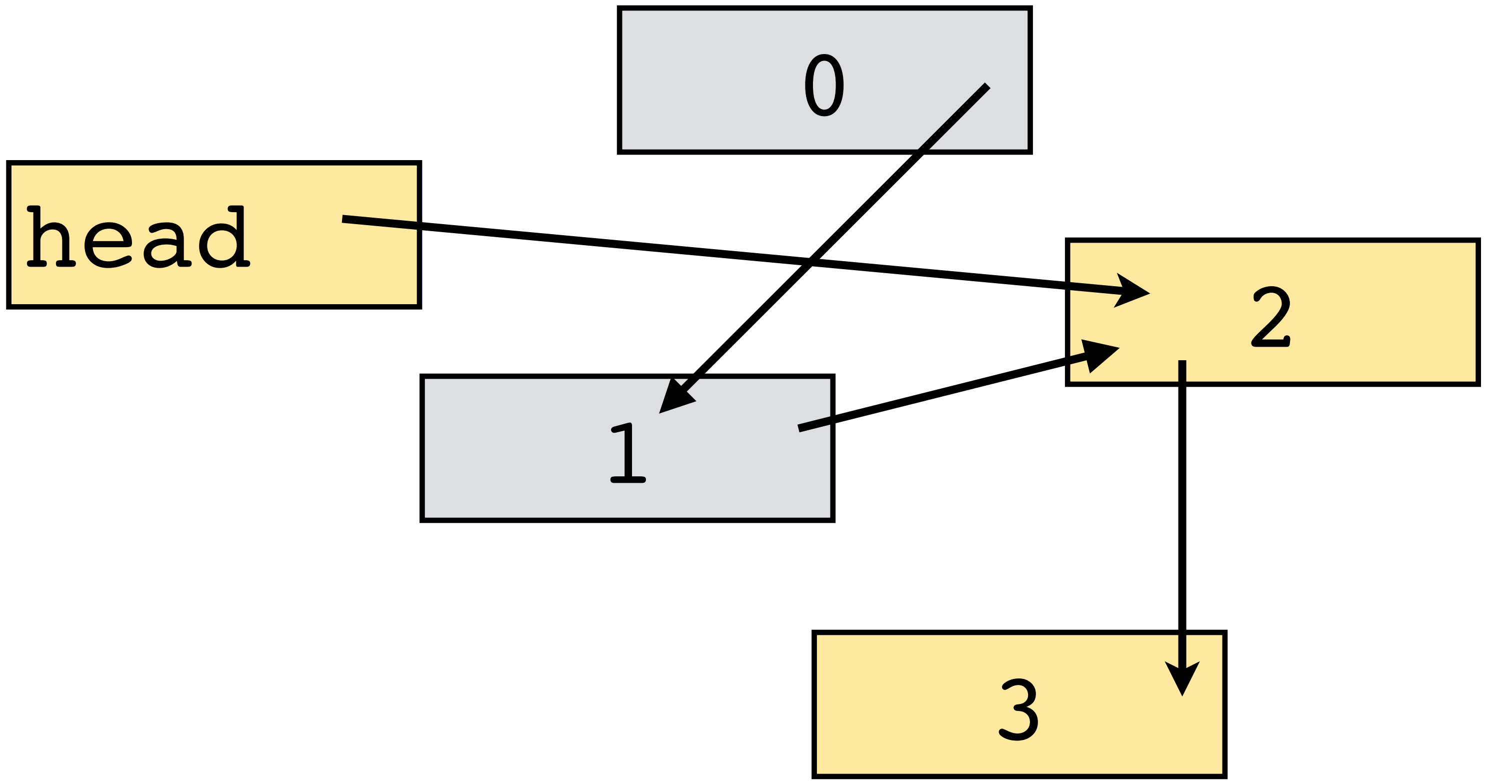
```
MyLinkedList L = new MyLinkedList();
L.add(0);
L.add(1);
L.add(2);
L.add(3);

L.remItemAt(1);
L.remItemAt(0);
```

Stack

Address	*Name*	*Contents*
10000	List L	&50000

Heap

Address	*Contents*
50000	head = &50040
50008	value=0, next=&50024
50024	value=1, next=&50040
50040	value=2, next=&50056
50056	value=3, next=null



Mark, Sweep

- Basic garbage collection algorithm
- Goal: find objects on the heap that are not referenced by any active object

- Maintain a list with a reference to all heap objects
  - Include a "referenced bit" with each object: 1=used, 0=lost

- Mark Phase:
  - Start from the root known object (e.g. top of stack)
  - Set referenced bit to 1 for every object it references

- Sweep Phase:
  - Step through list of all objects, delete anything not referenced

Garbage Collection

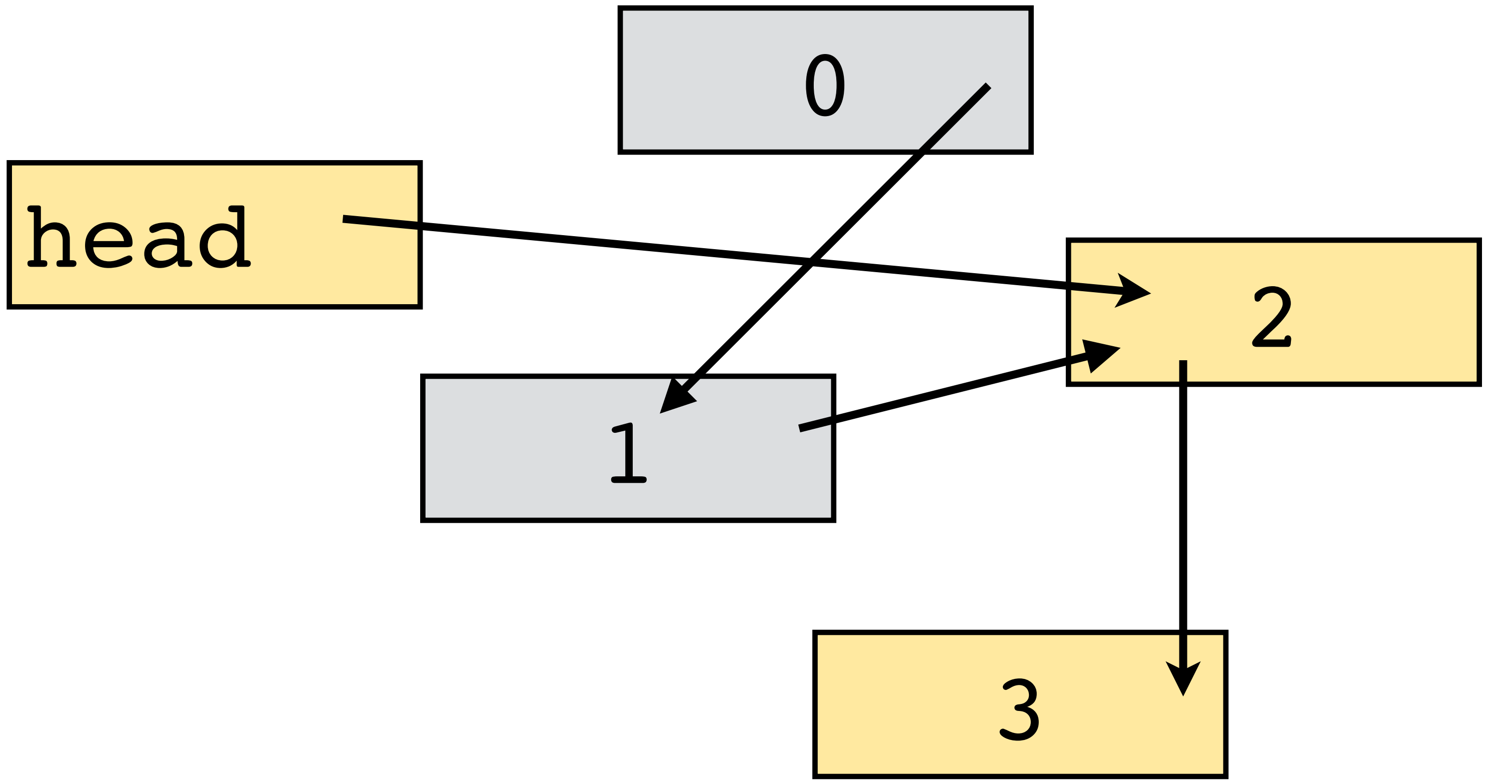
- The Java Run Time automatically tracks what objects are actively being used in memory
  - If no variable references something, then that object is "lost"---can be deleted

Stack

Address	*Name*	*Contents*
10000	List L	&50000

Heap

Address	*Reachable?*	*Contents*
50000		head = &50040
50008		value=0, next=&50024
50024		value=1, next=&50040
50040		value=2, next=&50056
50056		value=3, next=null



Benefits and Costs of GC

- Benefits:
  - No "memory leaks" from forgetting to free
  - Tighter control helps security

- Drawbacks:
  - May need to completely stop application while running garbage collection
  - Leads to unpredictable performance
  - Newer garbage collectors support parallelism
  - Just because there is a reference, doesn't mean it will be actively used again in the future