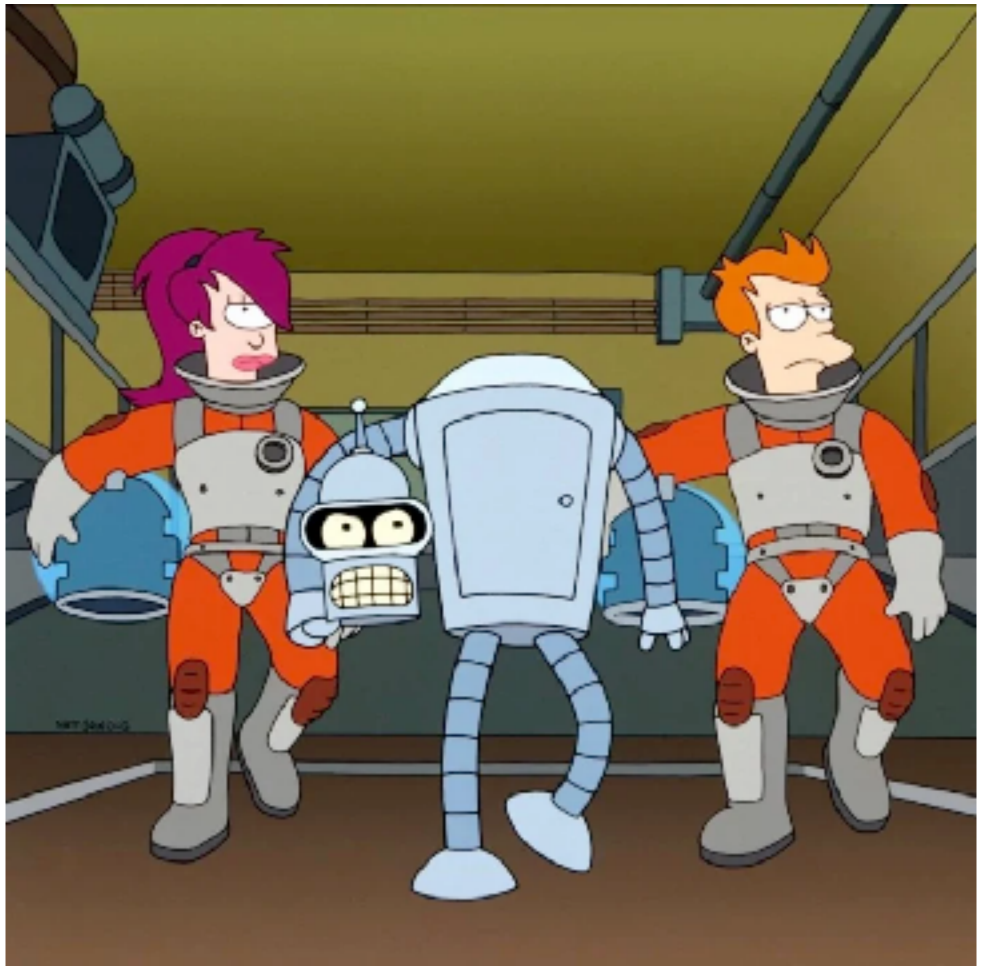
Object Paradigm and Terms

CSCI 4448/5448: Object-Oriented Analysis & Design

Computer Science is no more about computers than astronomy is about telescopes.

— Edsger Dijkstra

There are 10 kinds of people



- Those that understand binary and
- Those that don't

iClicker Quiz

What's This?

1000001

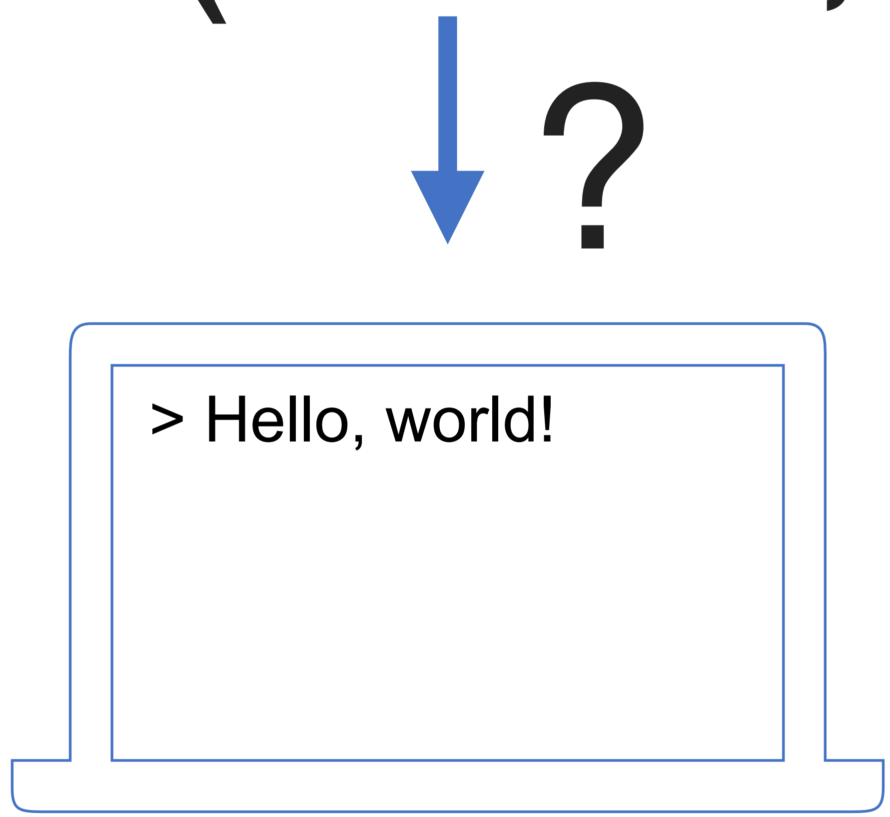
A. One million and one
B. 65
C. 'A'
D. 41

What's This?

1100 0001

- Binary number? What's the value?
- 65
- What about this?
- Hexadecimal number: 41_{16}
- ASCII 'A'
- A memory address
- A collection of boolean values

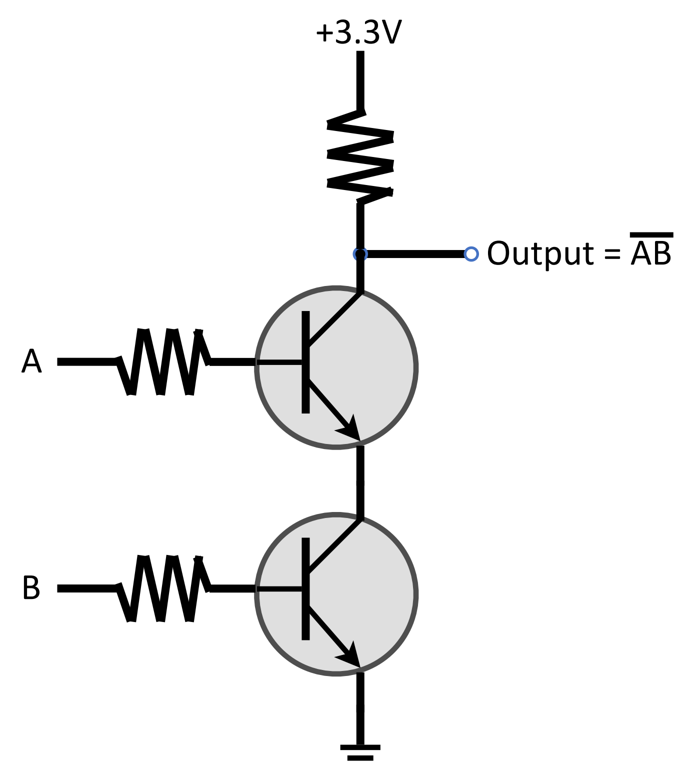
print(‘Hello, world!’)

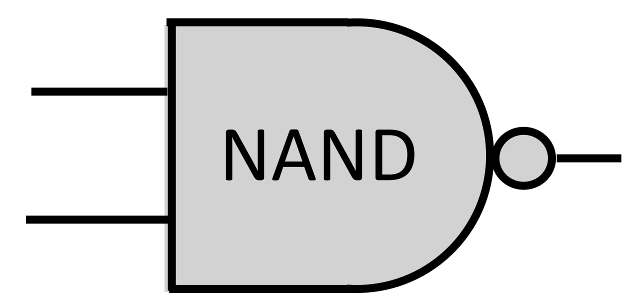


Vacuum Tubes

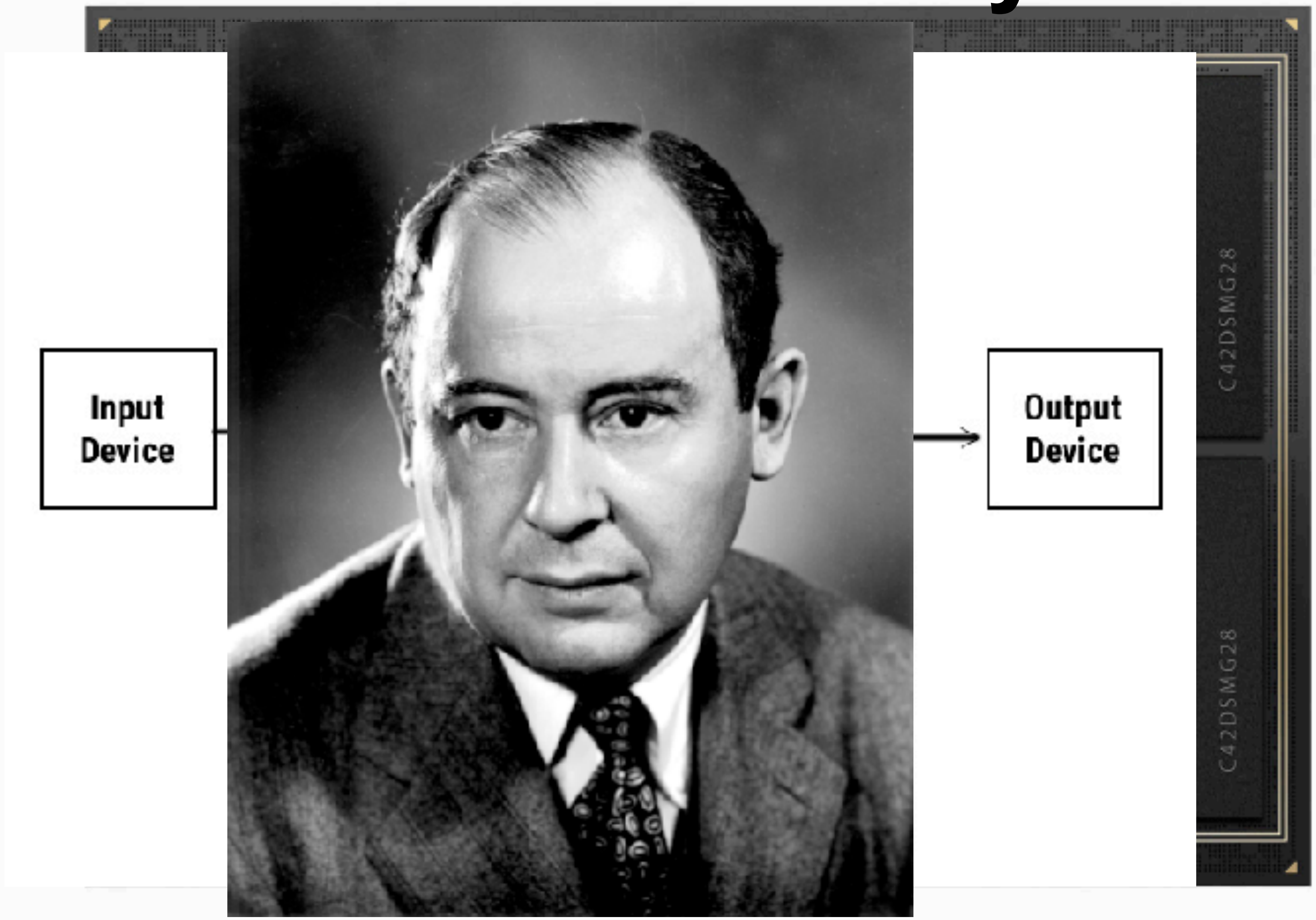
Transistors

Invented in 1947 at Bell Labs

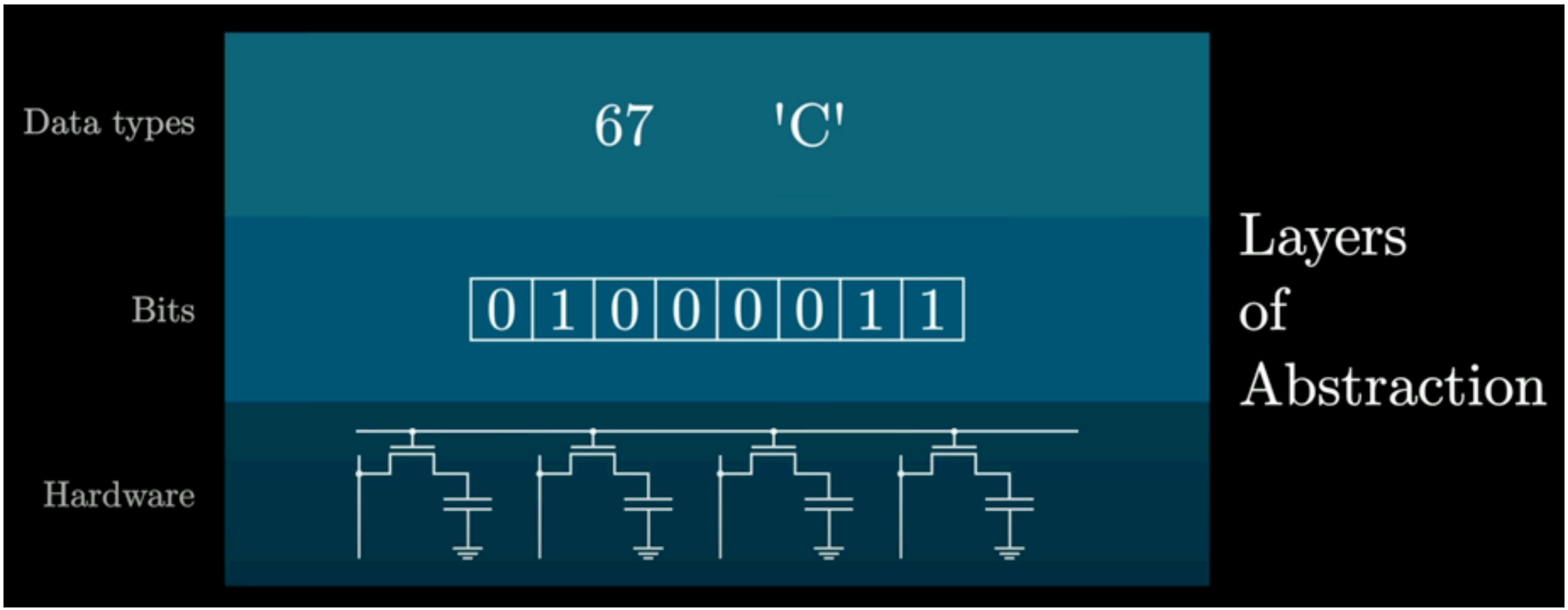




CPUs and Memory



Everything is built upon abstractions

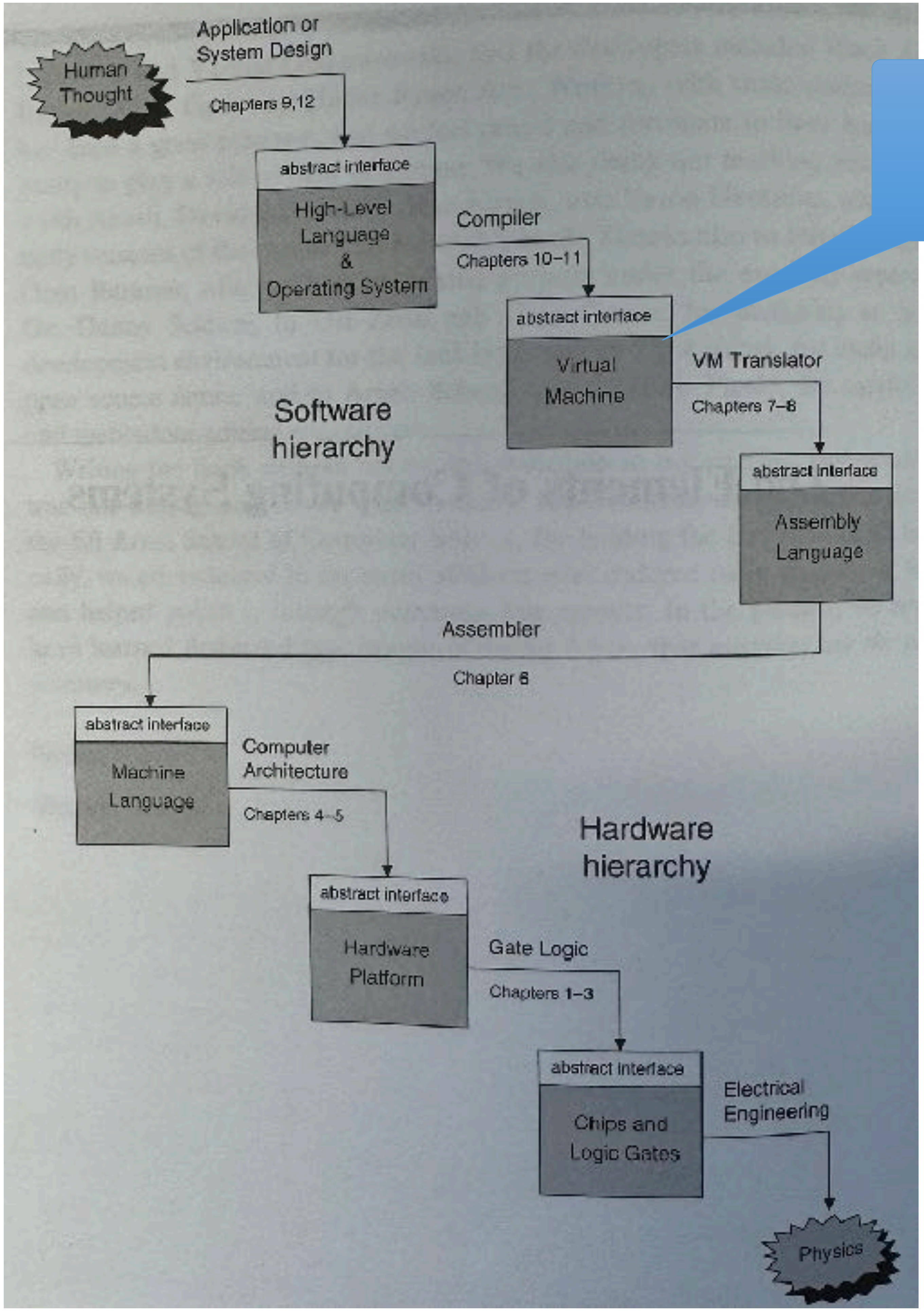


reference: https://www.youtube.com/watch?v=RQWpF2Gb-gU

Instruction Sets and Assembly Language

```
.global _start              // Provide program starting address to linker
.align 2

_start: mov X0, #1          // 1 = StdOut
        adr X1, helloworld  // string to print
        mov X2, #13         // length of our string
        mov X16, #4         // MacOS write system call
        svc 0               // Call linux to output the string

        mov     X0, #0       // Use 0 return code
        mov     X16, #1      // Service command code 1 terminates this program
        svc     0            // Call MacOS to terminate the program

helloworld:      .ascii  “Hello, World!\n"
```

Levels of Abstraction

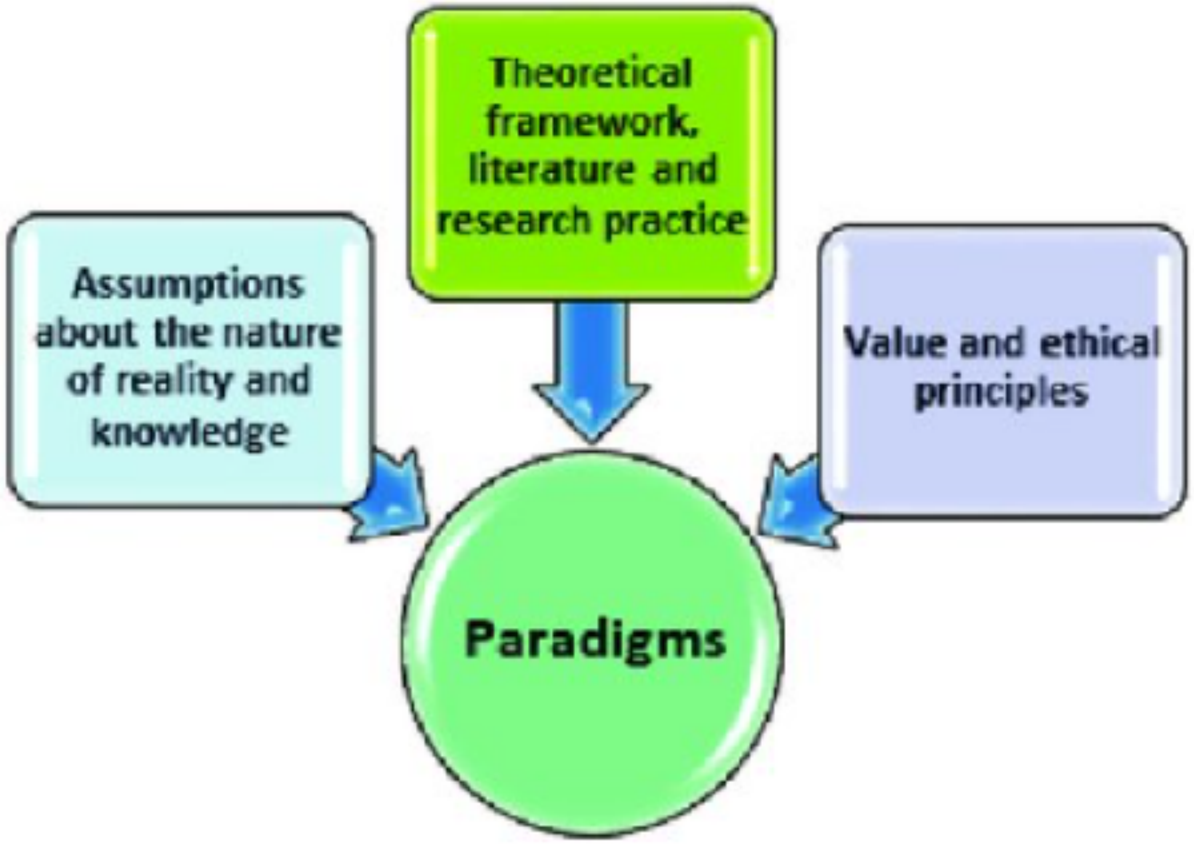


JVM

The Elements of Computer Systems by Nisan and Schocken

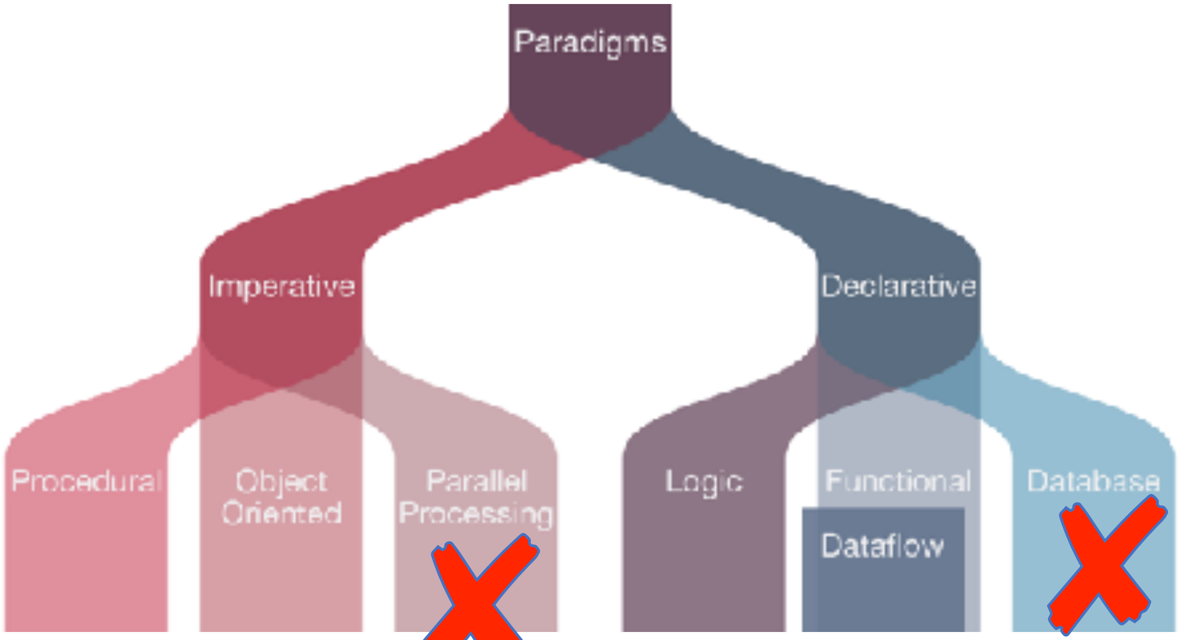
OO “Paradigm”

- Philosophical/theoretical framework for theories, laws, generalizations
- Patterns of thought and sets of supporting information



https://www.researchgate.net/publication/335396216_Selection_of_Research_Paradigms_in_English_Language_Teaching_Personal_Reflections_and_Future_Directions
https://www.merriam-webster.com/dictionary/paradigm

Design Methods



https://www.watelectronics.com/types-of-programming-languages-with-differences/

- **Structured Design/Programming**
  - “Think in terms of steps”
  - Example languages – C, Pascal, FORTRAN, Javascript, etc.
- **Object-Oriented Design/Programming**
  - “Think in terms of objects that do things”
  - Example languages – C++, C#, Java, Python, Objective-C, Scala
- **Functional Programming**
  - “Think in terms of functions and their composition”
  - Example languages – Mathematica, Lisp, Scala, Erlang, F#
- **Logic Programming**
  - “Concentrate on the what and not the how”
  - Example languages – Prolog

Procedural/Structural Programming

- Use of procedures/functions
- if-then-else branching constructs
- for, while, repeat looping constructs
- basically, no GOTO statements

```c
// Procedure to calculate the factorial of a number
int factorial(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

// Procedure to check if a number is prime
int is_prime(int n) {
    if (n <= 1) {
        return 0; // Not prime
    }
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            return 0; // Not prime
        }
    }
    return 1; // Prime
}
```

Functional Programming

- Recursion instead of looping structures (efficiency via tail recursion)
- Immutable data structures
- No side effects, so…?

This is a type declaration, just like "Int"

```
def performAddition(x: Int, y: Int): Int = x + y

def performSubtraction(x: Int, y: Int): Int = x - y

def performMultiplication(x: Int, y: Int): Int = x * y

def performArithmeticOperation(num1: Int, num2: Int, operation: String): Int = {
  operation match {
    case "addition" => performAddition(num1, num2)
    case "subtraction" => performSubtraction(num1, num2)
    case "multiplication" => performMultiplication(num1, num2)
  }
}
```

```
def performArithmeticOperation(num1: Int, num2: Int,
                               operation: (Int, Int) =>: Int): Int = {
  operation(num1, num2)
}
```

Declarative Programming

- Describe *what* you want the code to do, not *how*.

```
food_type(velveeta, cheese).
food_type(kraft, cheese).
food_type(taco_bell, tacos).
food_type(ice_cream, dessert).
food_type(twinkie, dessert).

flavor(sweet, dessert).
flavor(savory, tacos).
flavor(savory, cheese).

food_flavor(X, Y) :- food_type(X, Z), flavor(Y, Z).

?- food_flavor(velveeta, savory).
true.

?- food_flavor(taco_bell, sweet).
false.
```

```
sudoku(Rows) :-
    length(Rows, 9), maplist(same_length(Rows), Rows),
    append(Rows, Vs), Vs ins 1..9,
    maplist(all_distinct, Rows),
    transpose(Rows, Columns),
    maplist(all_distinct, Columns),
    Rows = [A,B,C,D,E,F,G,H,I],
    blocks(A, B, C), blocks(D, E, F), blocks(G, H, I).

blocks([], [], []).
blocks([A,B,C|Bs1], [D,E,F|Bs2], [G,H,I|Bs3]) :-
    all_distinct([A,B,C,D,E,F,G,H,I]),
    blocks(Bs1, Bs2, Bs3).

problem(1, [[_,_,_, _,_,_, _,_,_],
           [_,_,_, _,_,3, _,8,5],
           [_,_,1, _,2,_, _,_,_],

           [_,_,_, 5,_,7, _,_,_],
           [_,_,4, _,_,_, 1,_,_],
           [_,9,_, _,_,_, _,_,_],

           [5,_,_, _,_,_, _,7,3],
           [_,_,2, _,1,_, _,_,_],
           [_,_,_, _,4,_, _,_,9]]).
```

problem(1, Rows), *sudoku*(Rows).

Rows =

9	8	7	6	5	4	3	2	1
2	4	6	1	7	3	9	8	5
3	5	1	9	2	8	7	4	6
1	2	8	5	3	7	6	9	4
6	3	4	8	9	2	1	5	7
7	9	5	4	6	1	8	3	2
5	1	9	2	8	6	4	7	3
4	7	2	3	1	9	5	6	8
8	6	3	7	4	5	2	1	9

<u>**All**</u> Approaches Use Functional Decomposition

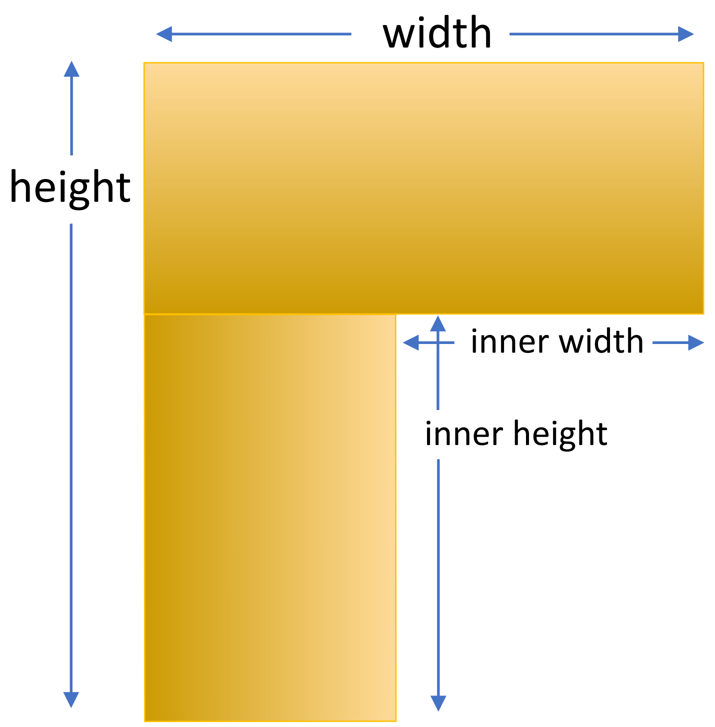
- **Decompose big problems into the functional steps required to solve it**
  - For a very big problem, simply break it down to small problems
    - then decompose small problems into smaller steps
- Goal is to **slice up the problems** until they are at a **level of granularity** that is **easy to solve in a couple of steps**
  - short functions are easier to understand
  - short functions are easier to reuse
  - short functions are easier to test

- If you have a large method, it can probably benefit from functional decomposition.

It gives us the opportunity to name the code!

```
int myLWidth = 5;
int myLInnerWidth = 3;
int myLHeight = 6;
int myLInnerHeight = 4;
```

```
int topHalfHeight = myLHeight - myLInnerHeight;
int topHalfArea = topHalfHeight * myLWidth;
int bottomHalfWidth = myLWidth - myLInnerWidth;
int topHalfArea = bottomHalfWidth * myLInnerHeight;
int myLArea = topHalfArea + topHalfArea;
```




```
int function getLArea(width, innerWidth, height, innerHeight) {
    int topHalfHeight = myLHeight - myLInnerHeight;
    int topHalfArea = topHalfHeight * myLWidth;
    int bottomHalfWidth = myLWidth - myLInnerWidth;
    int topHalfArea = bottomHalfWidth * myLInnerHeight;
    return topHalfArea + topHalfArea;
    }
```

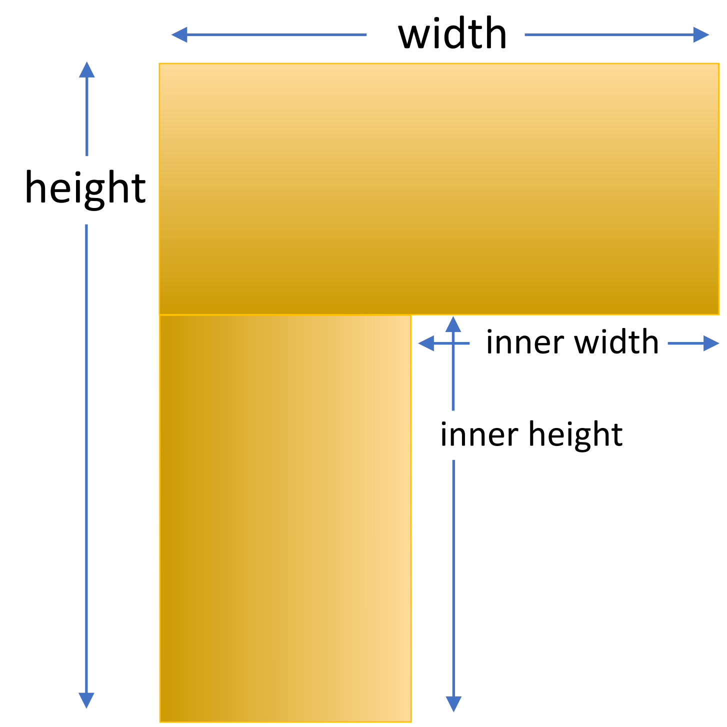
```
int myLArea = getLArea(myLWidth, myLInnerWidth, myLHeight, myLInnerHeight)
```

Procedural Approach

What if we had numerous rectangles?


```
int myRectWidth = 4;
int myRectHeight = 3;
int myRectArea = getRectangleArea(myRectWidth, myRectHeight)

int myRectWidth2 = 5;
int myRectHeight2 = 7;
int myRectArea = getRectangleArea(myRectWidth2, myRectHeight2)
```

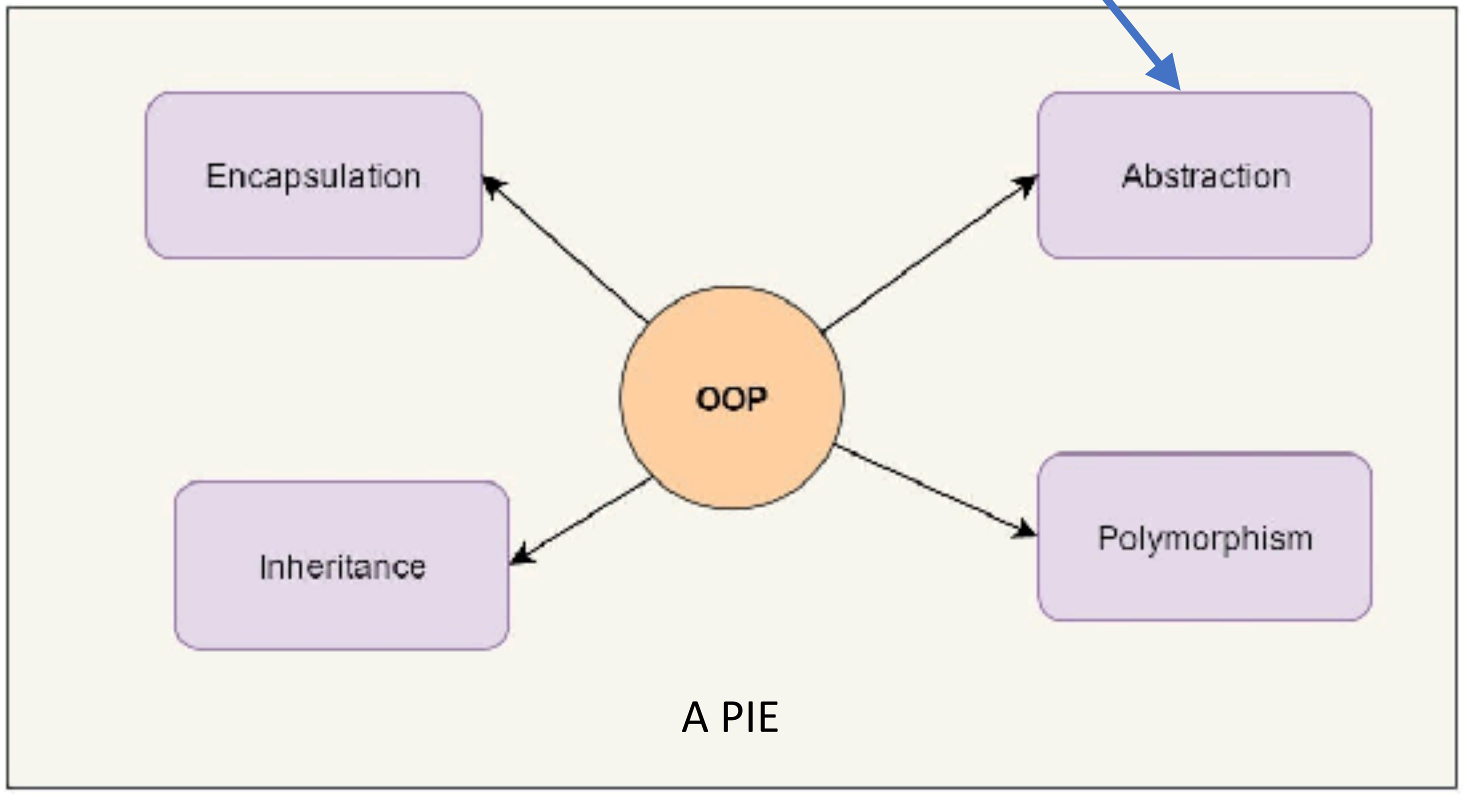
Or a complicated shape?




```
int myLWidth = 5;
int myLInnerWidth = 3;
int myLHeight = 6;
int myLInnerHeight = 4;
int myLArea = getLArea(myLWidth, myLInnerWidth,
                       myLHeight, myLInnerHeight)
```

What’s the problem?

poor use of **abstraction**



Four Pillars of Object Oriented Programming

Abstraction

C struct

```
struct Shape {
    int Height;
    int innerHeight;
    int Width;
    int innerWidth;
}
```

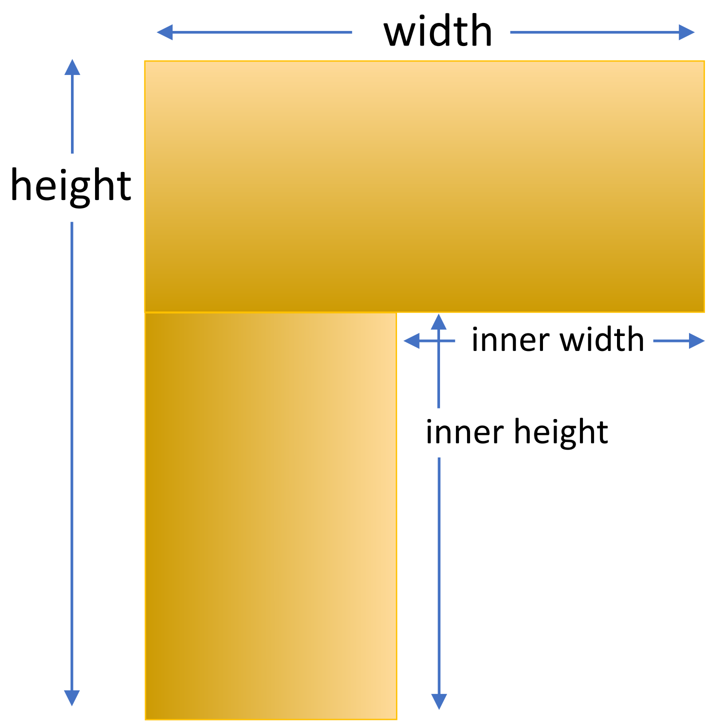
Java record

```
record Shape (int Width, int innerWidth,
              int Height, int innerHeight) {}
```

Abstraction Approach

```
Rectangle myRect = new Rectangle(4, 3);
int myRectArea = getRectangleArea(myRect.width, myRect.height)
```


```
LShape myLShape = new LShape(5, 3, 6, 4);
int myLArea = getLArea(myLShape.width, myLShape.innerWidth,
                       myLShape.height, myLShape.innerHeight)
```



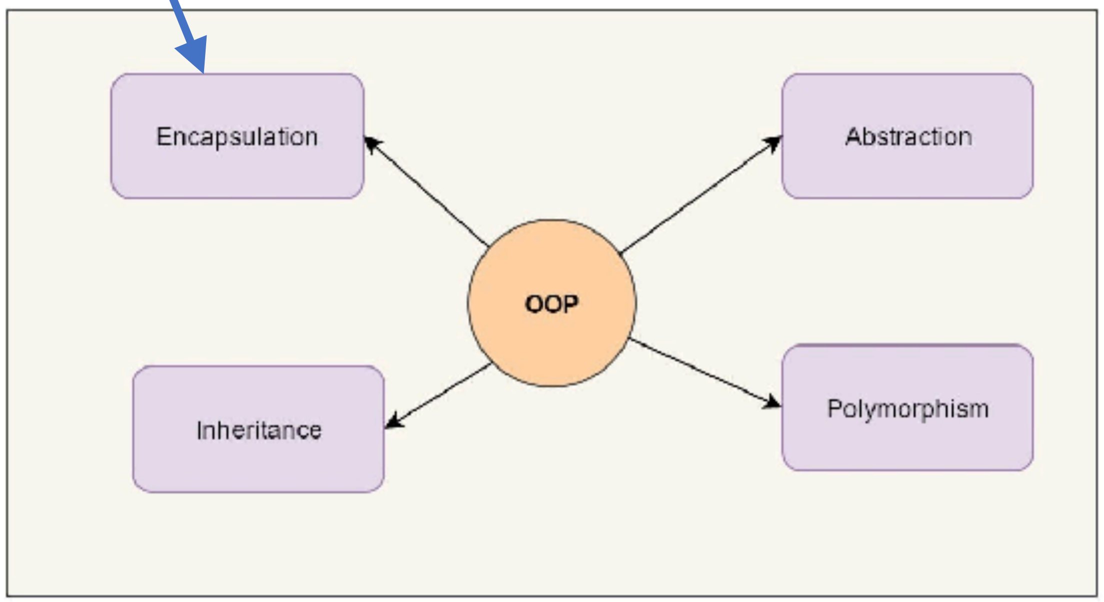
What's the problem?

- we need to know about the internals of our struct/record
- we need to match our function with our new abstraction

poor **encapsulation/information hiding**


```
Rectangle myRect = new Rectangle(4, 3);
int myRectArea = getRectangleArea(myRect.width, myRect.height)
```

Rectangle
Integer width Integer height
~~Integer getRectangleArea()~~ Integer area()



Four Pillars of Object Oriented Programming

```
Rectangle myRect = new Rectangle(4, 3);
int myRectArea = myRect.area()
```

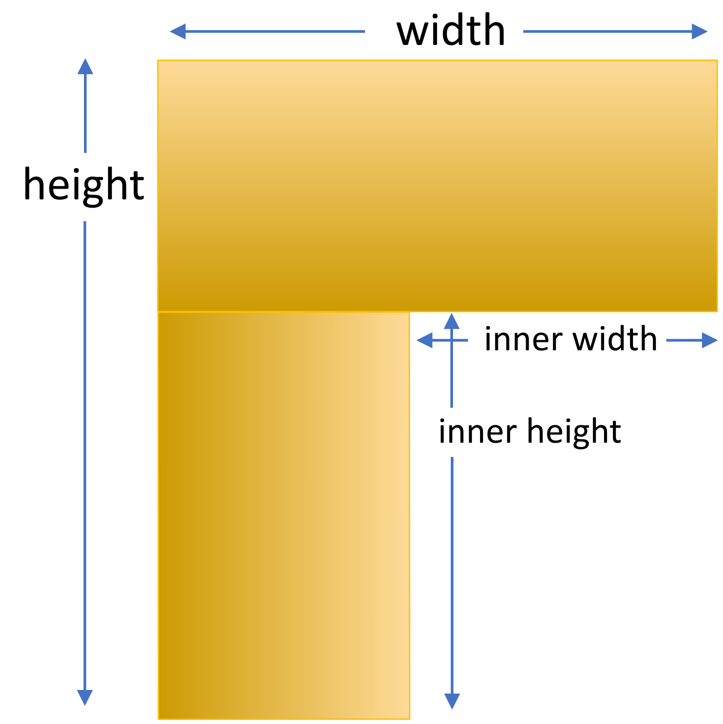
Abstraction & Encapsulation

- **Abstraction** refers to the **set of concepts that some entity provides (or exposes) for achieving a task or solve a problem**

- **Encapsulation** refers to a **set of language-level mechanisms or design techniques that hide implementation details** of a class, module, or subsystem from other classes, modules, and subsystems

And with a more complex abstraction…

```
LShape myLShape = new LShape(5, 3, 6, 4);
int myLArea = getLArea(myLShape.width, myLShape.innerWidth,
                       myLShape.height, myLShape.innerHeight)
```



LShape

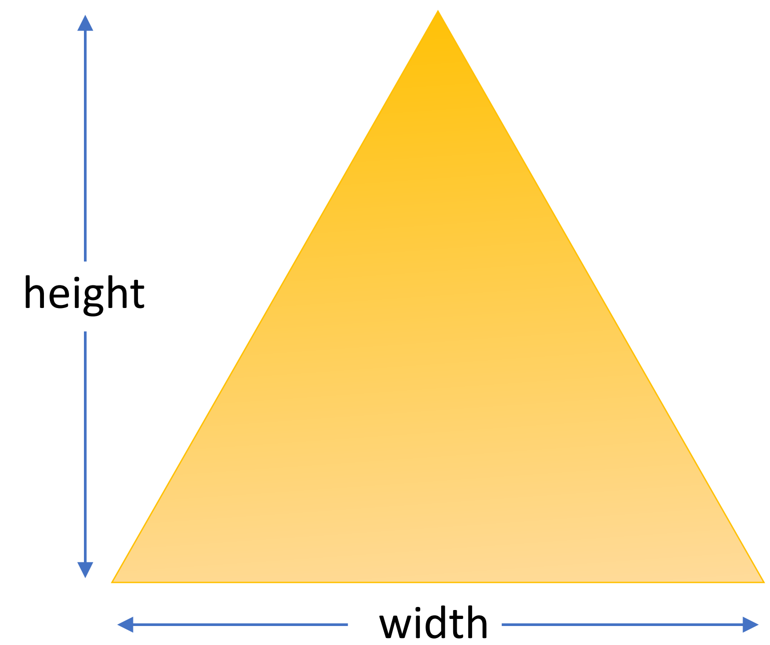
Integer width
Integer innerWidth
Integer height
Integer innerHeight

~~Integer getLArea()~~
Integer area()

```
LShape myLShape = new LShape(5, 3, 6, 4);
int myLShapeArea = myLShape.area()
```

What about a Triangle?

Triangle
Integer width Integer height
Integer area()

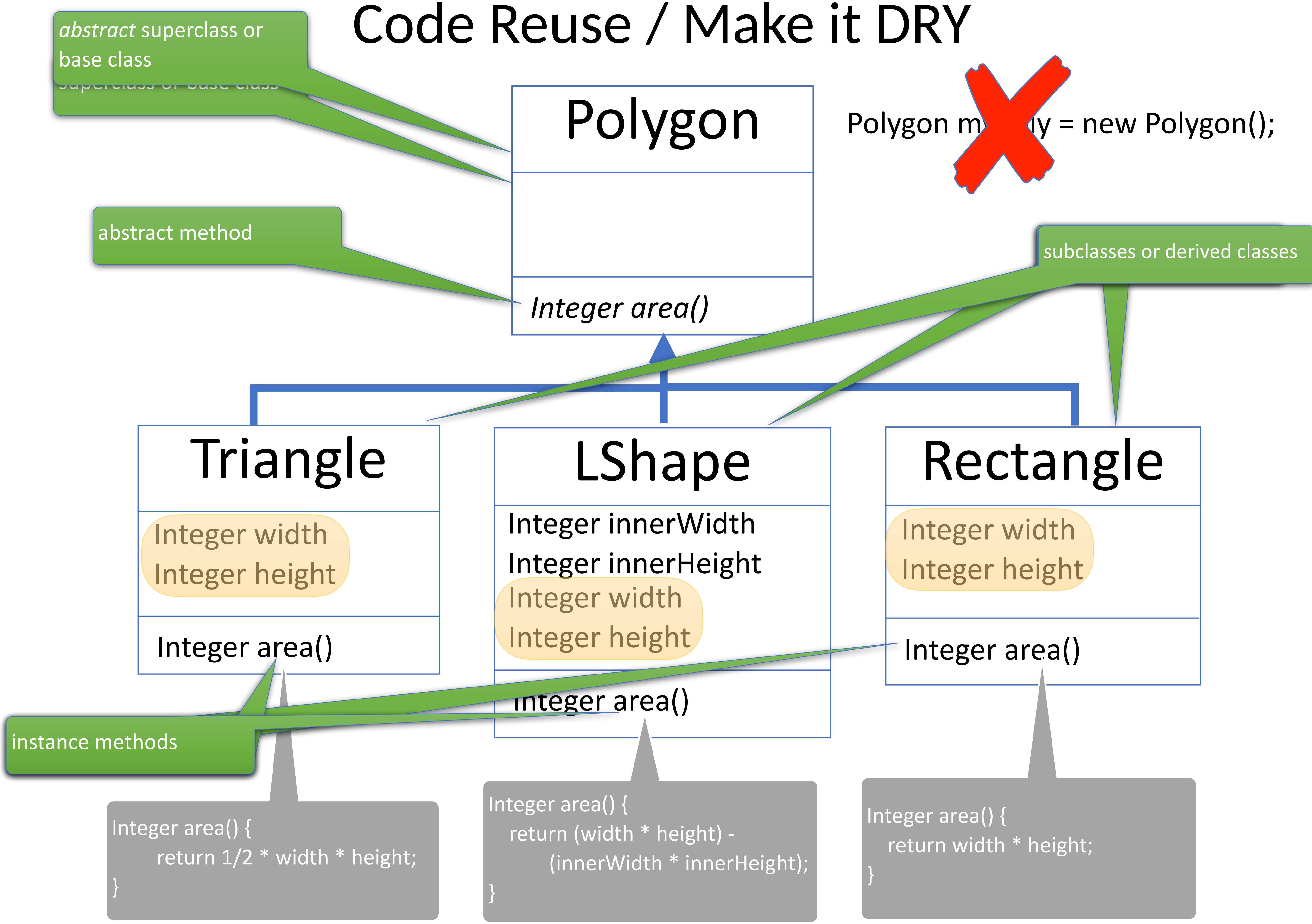


LShape
Integer width Integer innerWidth Integer height Integer innerHeight
Integer area()

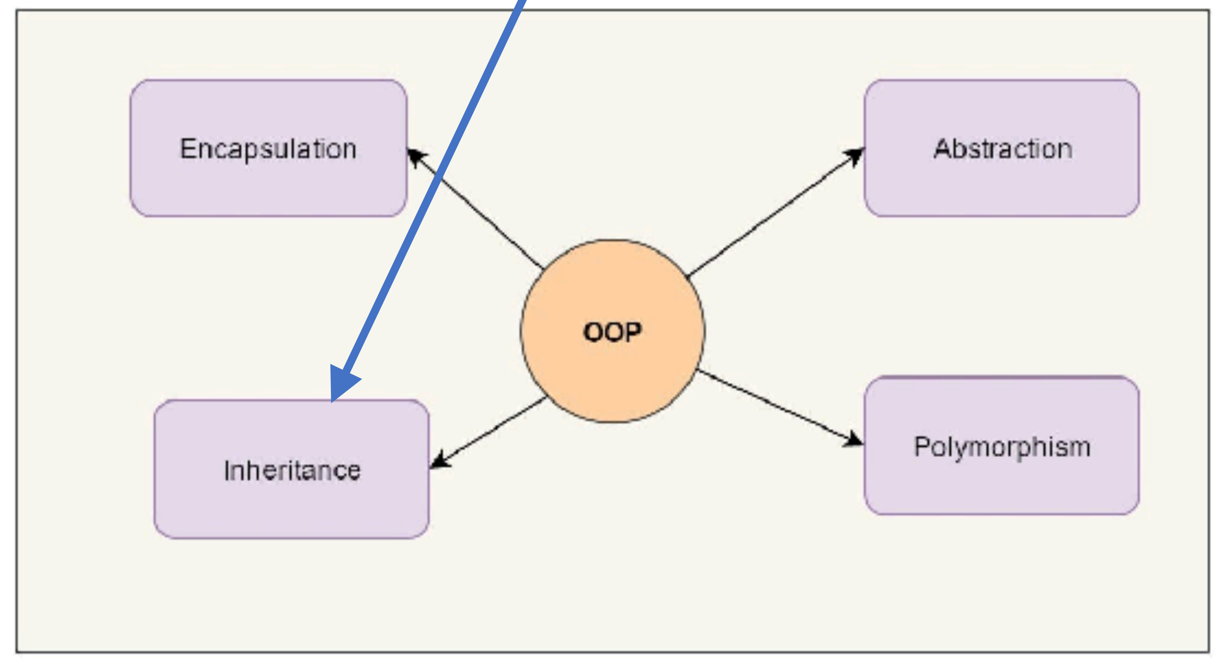
Rectangle
Integer width Integer height
Integer area()

What’s the problem?

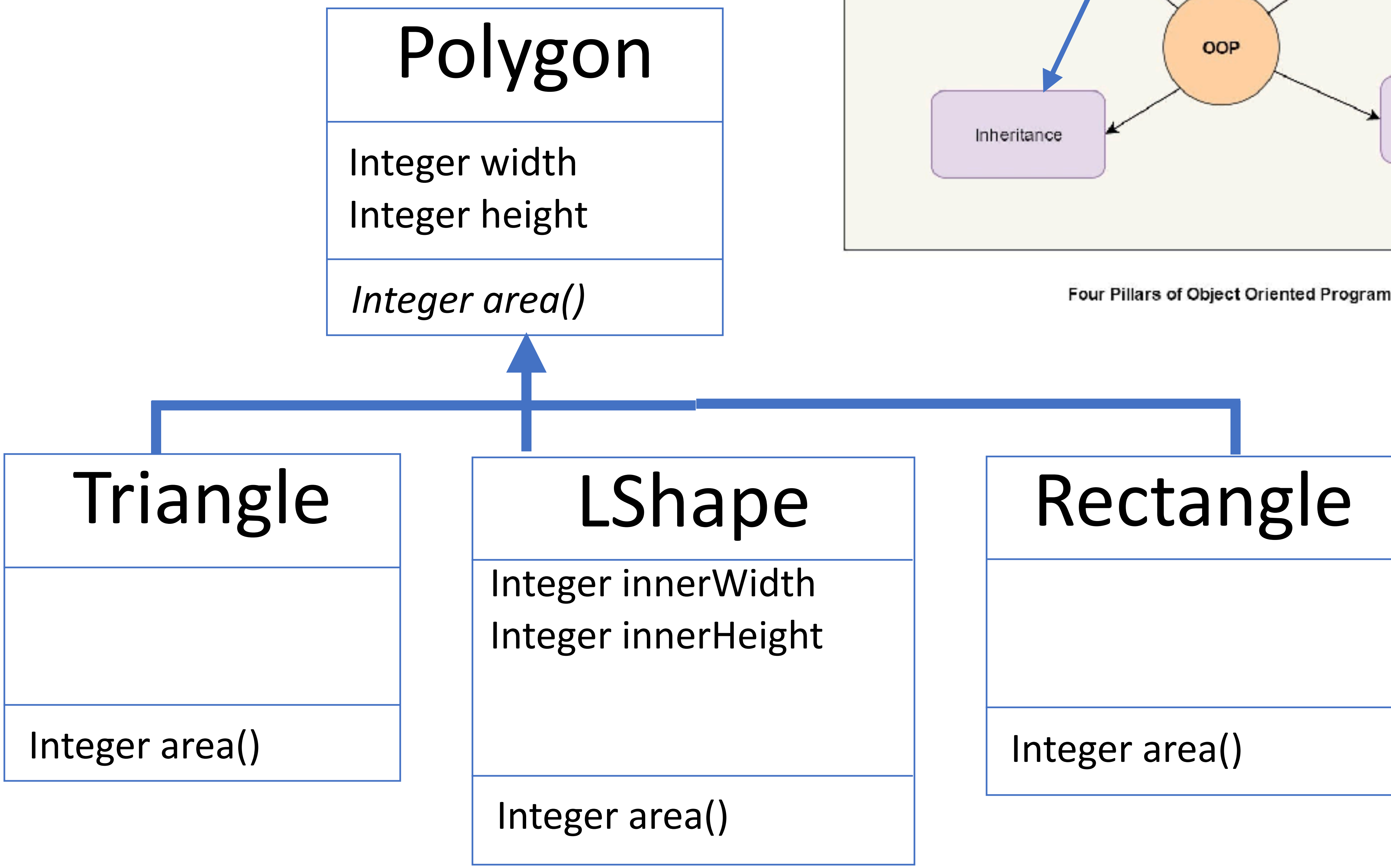
Code Reuse / Make it DRY



Code Reuse with Inheritance



Four Pillars of Object Oriented Programming



Polymorphism

Many

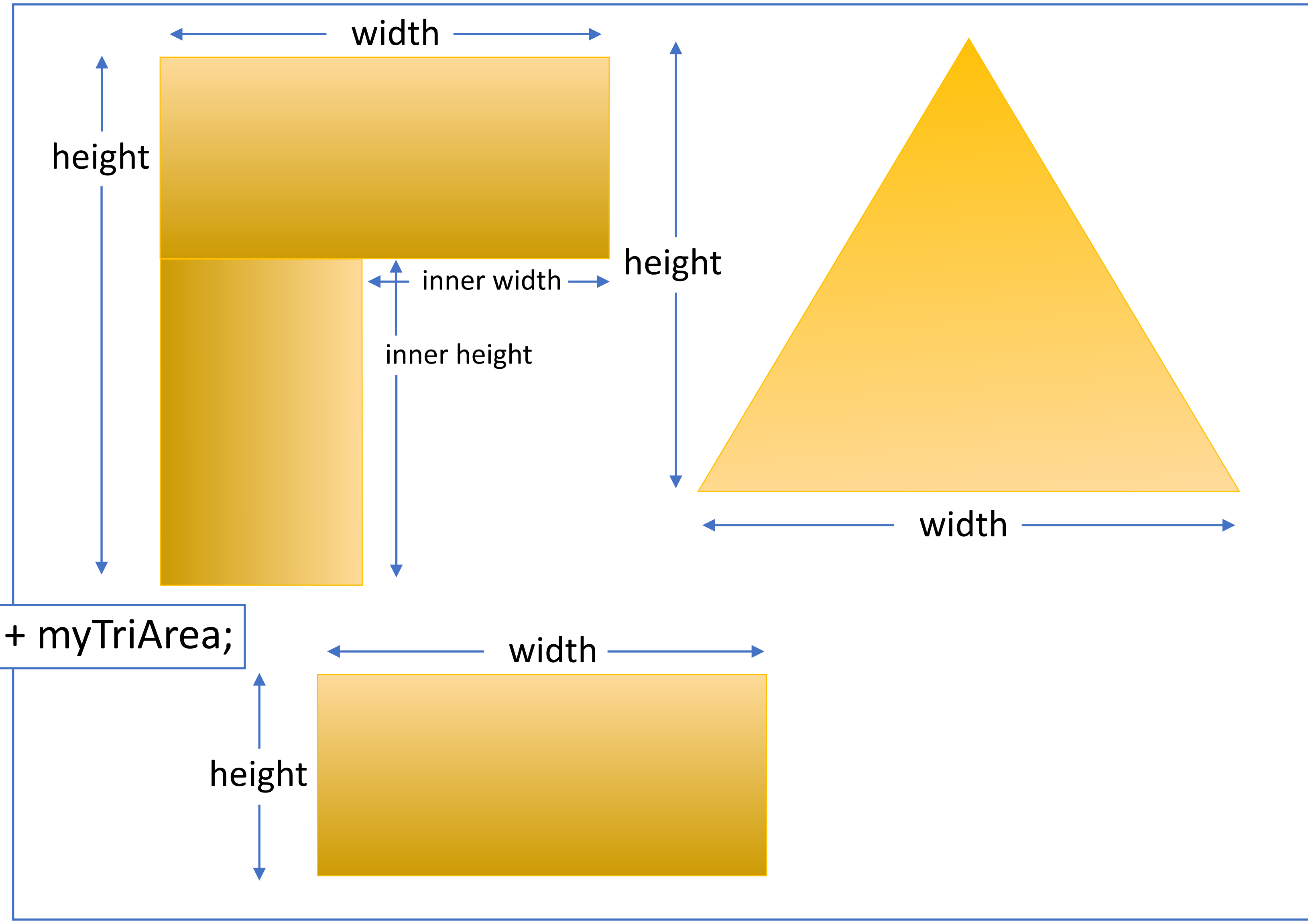
Shapes

```
Rectangle myRect = new Rectangle(4, 3);
int myRectArea = myRect.area();
```

```
LShape myLShape = new LShape(5, 3, 6, 4);
int myLShapeArea = myLShape.area();
```

```
Triangle myTri = new Triangle(5, 4);
int myTriArea = myTri.area();
```

```
int totalArea = myRectArea  + myLShapeArea  + myTriArea;
```




```
List<Shape> myShapes = List.of(new Rectangle(4, 3), new LShape(5, 3, 6, 4), new Triangle(5, 4));
int totalArea = 0;
for (Shape currentShape : myShapes) {
   totalArea += currentShape.area();
}
```

This method has many shapes (multiple implementations)

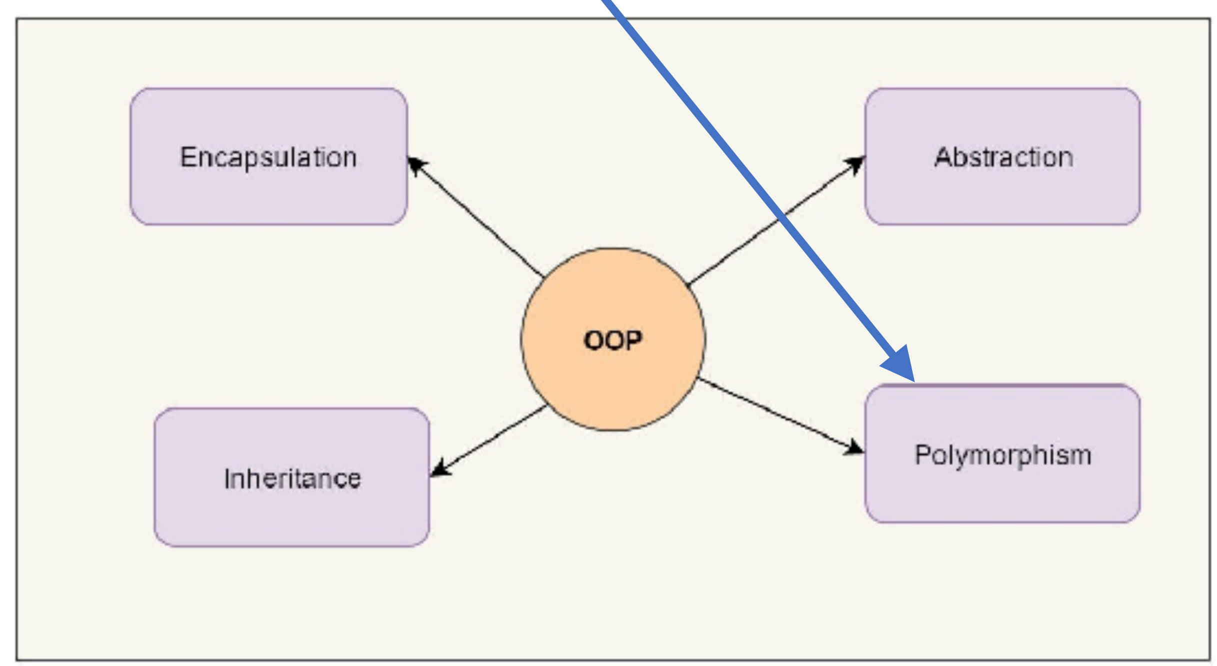
Polymorphism

Many

Shapes

```
List<Shape> myShapes = List.of(new Rectangle(4, 3), new LShape(5, 3, 6, 4), new Triangle(5, 4));
int totalArea = 0;
for (Shape currentShape : myShapes) {
   totalArea += currentShape.area();
}
```

```
int totalArea(List<Shape> myShapes)
   int totalArea = 0;
   for (Shape currentShape : myShapes) {
      totalArea += currentShape.area();
   }
   return totalArea;
}
```



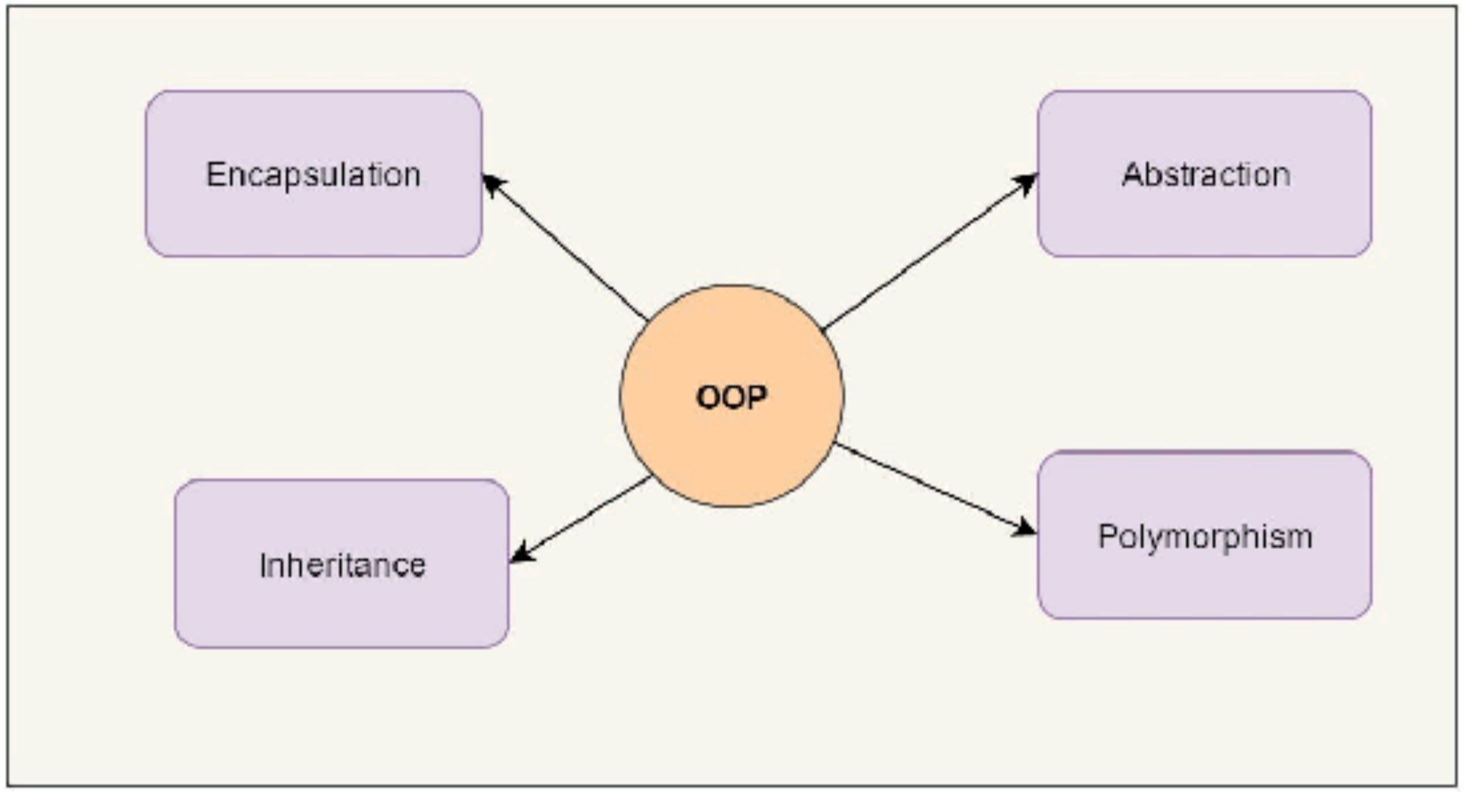
Four Pillars of Object Oriented Programming

Assumed Vocabulary

- Class
- Object
- Interface
- Superclass or Base Class
- Subclass or Derived Class
- Instance variable
- Instance method
- Method signature
- Access modifiers (private, public)

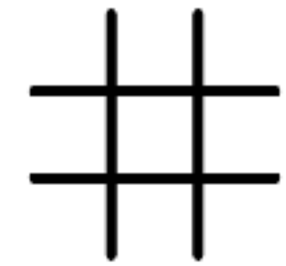
Why do these problems exist?

- poor use of **abstraction**
- poor **encapsulation** (a.k.a. **information hiding**)
- poor **modularity**



Four Pillars of Object Oriented Programming

Abstraction Example: TicTacToe



[[‘X’, ‘ ‘, ‘O’],
[‘X’, ‘ ‘, ‘O’],
[‘X’, ‘ ‘, ‘O’]]

Array of Arrays:

```
public class TicTacToe {
    private List<List<String>> board = new ArrayList<>();

    public void makeMove(String playerCharacter, Integer row, Integer column) {
        List<String> row = board.get(row);
        row.set(column, playerCharacter);
```

Array:

```
public class TicTacToe {
    private List<String> board = new ArrayList<>();

    public void makeMove(String playerCharacter, Integer row, Integer column) {
        int index = row * 3 + column;
        board.set(index, playerCharacter);
```

[‘X’, ‘ ‘, ‘O’, ’X’, ‘ ‘, ‘O’, ‘X’, ‘ ‘, ‘O’]

Map or Dictionary:

```
public class TicTacToe {
    private Map<String, String> board = new HashMap<>();

    public void makeMove(String playerCharacter, Integer row, Integer column) {
        String key = "R" + row.toString() + "C" + column.toString();
        board.put(key, playerCharacter);
```

{‘R0C0’: ‘X’, ‘R1C1’: ‘ ‘ , ... }

Understanding the Benefits of Using Classes

- **Model and solve complex real-world problems:** You'll find many situations where the objects in your code map to real-world objects. This can help you think about complex problems, which will result in better solutions to your programming problems.

- **Reuse code and avoid repetition:** You can define hierarchies of related classes. The base classes at the top of a hierarchy provide common functionality that you can reuse later in the subclasses down the hierarchy. This allows you to reduce code duplication and promote code reuse.

- **Encapsulate related data and behaviors in a single entity:** You can use Python classes to bundle together related attributes and methods in a single entity, the object. This helps you better organize your code using modular and autonomous entities that you can even reuse across multiple projects.

- **Abstract away the implementation details of concepts and objects:** You can use classes to abstract away the implementation details of core concepts and objects. This will help you provide your users with intuitive interfaces (APIs) to process complex data and behaviors.

- **Unlock polymorphism with common interfaces:** You can implement a particular interface in several slightly different classes and use them interchangeably in your code. This will make your code more flexible and adaptable.

Sudoku

1							4	
		6	7					
2	8	4			6		7	3
	7							
5	4	9		1			8	7
			2	5			6	
	1		3					2
	6	3		2	9		1	
			5	6				

Structural/Procedural Version Sudoku

```
column_names = 'ABCDEFGHI'
row_names = '123456789'
row_box_groupings = ('ABC','DEF','GHI')
col_box_groupings = ('123','456','789')

def box_groupings(column_names, row_names):
    return [(row, column) for row in row_names for column in column_names]

puzzle_input_string = '..3.6.7..9...78'
def create_puzzle(sudoku_input_string)
    """return dictionary_representing_sudoku_puzzle”""

def cell_addresses(puzzle_string):
    pass
```

Structural/Procedural Approach: Main Problems

- It creates designs **centered around a “main program”**

  This program is in control and knows all of the details about what needs to be done and all of the details about the program’s data structures

- It creates designs that **do not respond well to change requests**

  These programs are **not well modularized** and so a change request often requires modification of the main program; a minor change in a data structure, for example, might cause impacts throughout the entire main program

Why should we care?

One constant in software development: Change

- Research has shown that many bugs originate with changes to the code.
- We don't want to get overwhelmed by change requests.
- We need designs that are *resilient* to change;
- Indeed, we want software that is "designed" to accommodate change in a straightforward manner

Evolving to Objects

```
driver = ('Left', 5, 12)

section = driver[0]

driver = {‘section’: ’Left', ‘row’: 5, ’seat’: 12}

section = driver[‘section’]
```

Python Object

```python
class StudentLocation:

    def __init__(self, row, seat):
        self.row = row
        self.seat = seat
```

```python
student = StudentLocation('Right', 3, 5)
student.seat
```

With objects we can then add behavior

```python
student.moveLeft()
student.moveUpARow()
```

Java Object

```
class StudentLocation {
    Integer row;
    Integer seat;

    public StudentLocation(Integer aRow, Integer aSeat) {
        this.row = aRow;
        seat = aSeat;
    }
}
```

```
StudentLocation student = StudentLocation(3, 5);
student.seat;
```

OO Sudoku

What objects might help us here?

- SudokuPuzzle?
- Cell?
- Candidates?
- Group?
- Strategy?

Only three hard problems in computer science

Abstraction Example

```
value = memcache.get(key)
if (! value) {
  value = database.get(key)
  memcache.put(key, value)
}
...value...
```

Repeated many times

```
value = cache.get(key)
```

```
value = datastore.get(key)
```

The Problems with **just** Functional Decomposition

- weak cohesion
- tight coupling

```
public void process_records(List<Record> records) {
  // sort records, update values in records, print records,
  // archive records and log each operation as it is performed …
}

public void do_stuff(List<Record> records) {

}
```

Cohesion

- Cohesion refers to “how closely the operations in a routine are related”
  - A simplification is to say “*we want this method to do just one thing*” or “*we want this module to deal with just one thing*”
- We want our code to exhibit **strong cohesion** (a.k.a. highly cohesive)
  - methods: the method performs one operation
  - classes: the class achieves a fine-grain design or implementation goal
  - packages: the package achieves a medium-grain design goal
  - subsystems: this subsystem achieves a coarse-grain design goal
  - system: the system achieves all design goals and meets its requirements

Ref: Code Complete 2 by Steve McConnell; Microsoft Press, 2004

Coupling

- Coupling refers to “the strength of a connection between two routines”
  - It is a complement to cohesion
    - weak cohesion implies strong coupling
    - strong cohesion implies loose coupling
- With strong or tight coupling, a single change in one method or data structure will cause **ripple effects**, that is, additional changes in other parts of the system – often unwanted side effects from change
- We want systems with parts that are **highly cohesive** and **loosely coupled**

Ref: Code Complete 2 by Steve McConnell; Microsoft Press, 2004

The Object-Oriented Paradigm

- OO Analysis & Design is centered around the concept of an **object**
  - It produces systems that are **networks of objects** *collaborating* to **fulfill the responsibilities** (requirements) of the system
- Objects are conceptual units that combine both data and behavior
  - The **data** of an object is referred to by many names:

    attributes, properties, instance variables, etc.
  - The **behavior** of an object is defined by its **set of methods**
  - Methods and attributes are **members** of an object (or class)
- Objects **inherently know what type they are**.
  - Its attributes **allows it to keep track of its state**.
  - Its methods **allow it to function properly**.

Return to Slide 41

Object Responsibilities

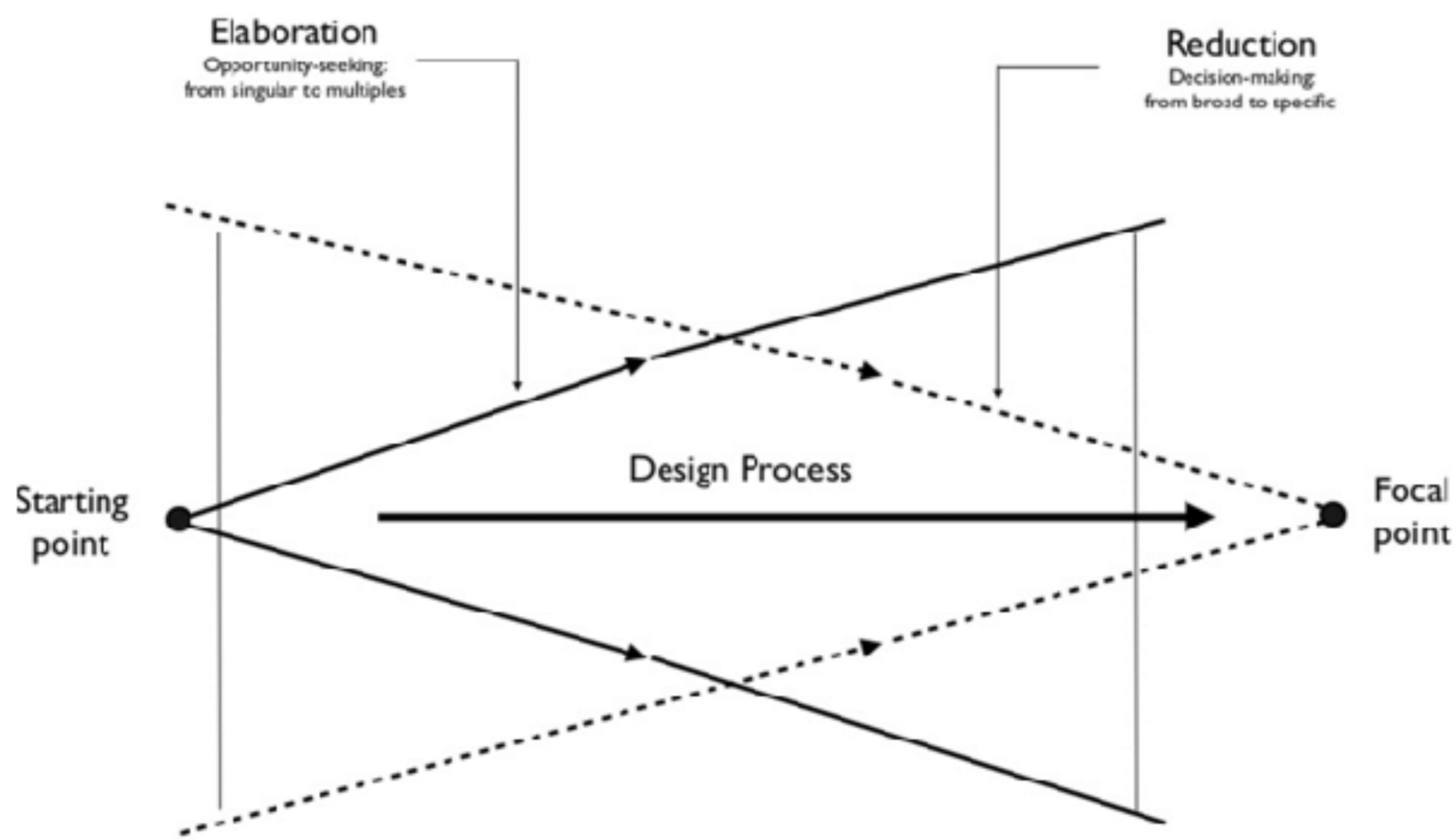
- In OO Analysis and Design, it is best to think of an object as “**something with responsibilities**”
  - As you perform analysis (What’s the problem?), you discover responsibilities that the system must fulfill
  - You will eventually find “homes” for these responsibilities in the objects you design for the system; indeed this process can help you “discover” objects needed for the system
  - The problem domain will also provide many candidate objects to include in the system
  - **This is an example of moving from a conceptual perspective to the specification and implementation perspectives**

Objects and Design Perspectives

- Conceptual — a set of responsibilities
- Specification — a set of methods
- Implementation — a set of code and data
- Unfortunately, OO A&D is often taught only at the implementation level
  - if previously you have used OO programming languages without doing analysis and design up front, then you've been operating only at the implementation level
    - "jumping into code"
  - as you will see, there are great benefits from starting with the other levels first

Benefits of Thinking Conceptually

- Broad thinking in the early phases of **Progressive Elaboration** on a design
- Avoidance of **Premature Optimization** away from possible design choices
- In Object terms – focusing on responsibilities of the system and its elements



Aside: Sketching (defined broadly) is a great way to explore many different ideas quickly…

- From Sketching User Experiences, Buxton, 2007, Morgan Kaufmann

```
grid = '..3.2.6..9..3.5..1..18.64....81.29..7.......8..67.82....26.95..8..2.3..9..5.1.3..'
```

Sodoku Structured (non-OO) Solution

```
column_names = 'ABCDEFGHI'
row_names = '123456789'
possible_values = '123456789'

def column_names(puzzle_string):
    puzzle_size = int(math.sqrt(len(puzzle_string)))
    # 65 is the ASCII number for a capital A
    return ''.join(chr(65+col_number) for col_number in range(puzzle_size))

def row_names(puzzle_string):
    puzzle_size = int(math.sqrt(len(puzzle_string)))
    return ''.join(str(row_number+1) for row_number in range(puzzle_size))

def ordered_cell_addresses(puzzle_string):
    addresses = []
    for col in column_names(puzzle_string):
        for row in row_names(puzzle_string):
            addresses.append(col + row)
    return addresses
    # return cross(column_names, row_names)

def create_ordered_values_from_puzzle_string(puzzle_string):
    return [(value if value != '.' else possible_values) for value in puzzle_string]

def create_puzzle(sudoku_puzzle_string):
    addresses = ordered_cell_addresses(sudoku_puzzle_string)
    values = create_ordered_values_from_puzzle_string(sudoku_puzzle_string)
    return dict(zip(addresses, values))

def display_puzzle_simple(puzzle, row_names, column_names):
    width = 1 + max(len(s) for s in puzzle.values())
    print('    ' + ''.join(col_name.center(width, ' ') for col_name in list(column_names)))
    for row_name in row_names:
        row_values = (puzzle[col_name+row_name].center(width) for col_name in column_names)
        print(row_name + ' |' + ''.join(row_values))
```

Sodoku Structured (non-OO) Solution

```python
def eliminate(values):
    eliminate_singles(values)
    eliminate_pairs(values)
    eliminate_triples(values)

    return values


def eliminate_singles(values):
    """Eliminate values from peers of each box with a single value.

    Go through all the boxes, and whenever there is a box with a single value,
    eliminate this value from the set of values of all its peers.

    Args:
        values: Sudoku in dictionary form.
    Returns:
        Resulting Sudoku in dictionary form after eliminating values.
    """
    for box, possibleValues in values.items():
        if len(possibleValues) == 1:
            for peer in peers[box]:
                values[peer] = values[peer].replace(possibleValues, "")
    return values


def eliminate_pairs(values):
    for unit in unit_list:
        pairs = [[box1, box2] for box1 in unit for box2 in unit if box1 != box2]
        for pair in pairs:
            if values[pair[0]] == values[pair[1]] and len(values[pair[0]]) == 2:
                for peer in unit:
                    if peer != pair[0] and peer != pair[1]:
                        print("Found pair: " + str(pair) + ", in unit: " + str(unit))
                        for curr_digit in values[pair[0]]:
                            values[peer] = values[peer].replace(curr_digit, '')

def eliminate_triples(values):
    for unit in unit_list:
        triples = [[box1, box2, box3] for box1 in unit for box2 in unit for box3 in unit if (box1 != box2) and (box1 != box3) and (box2 != box3)]
        for triple in triples:
            if values[triple[0]] == values[triple[1]] and (values[triple[0]] == values[triple[2]]) and (values[pair[0]]) == 3:
                for peer in unit:
                    if peer != triple[0] and peer != triple[1] and peer != triple[2]:
                        print("Found triple: " + str(triple) + ", in unit: " + str(unit))
                        for curr_digit in values[triple[0]]:
                            values[peer] = values[peer].replace(curr_digit, '')
```

Sodoku OO Solution

```
class Sudoku(GridPuzzle):

    def __init__(self, puzzle_string):
        super().__init__(puzzle_st
        self.box_group_size = int(
        self.column_boundaries = s
        self.row_boundaries = self
        self.box_groups = self.cre
```

```
def test_solve_expert_16x16_puzzle(self):
    puzzle = Sudoku(self.expert_16x16)
    puzzle.display()

    solved_puzzle = puzzle.search()
    solved_puzzle.display()
```

```
class Reducing_Group(Group):

    def search_and_reduce_exclusions(self):
        """This method will modify the cells in the group, if exclusive cells are found."""
        self.check_consistency()
        candidate_cell_map = dict()    # Here we keep track of each candidate and which cells it
appears in
        for cell in self.cells:
            for candidate in cell.candidates:
                if candidate not in candidate_cell_map:
                    candidate_cell_map[candidate] = []
                candidate_cell_map[candidate].append(cell)
…
```

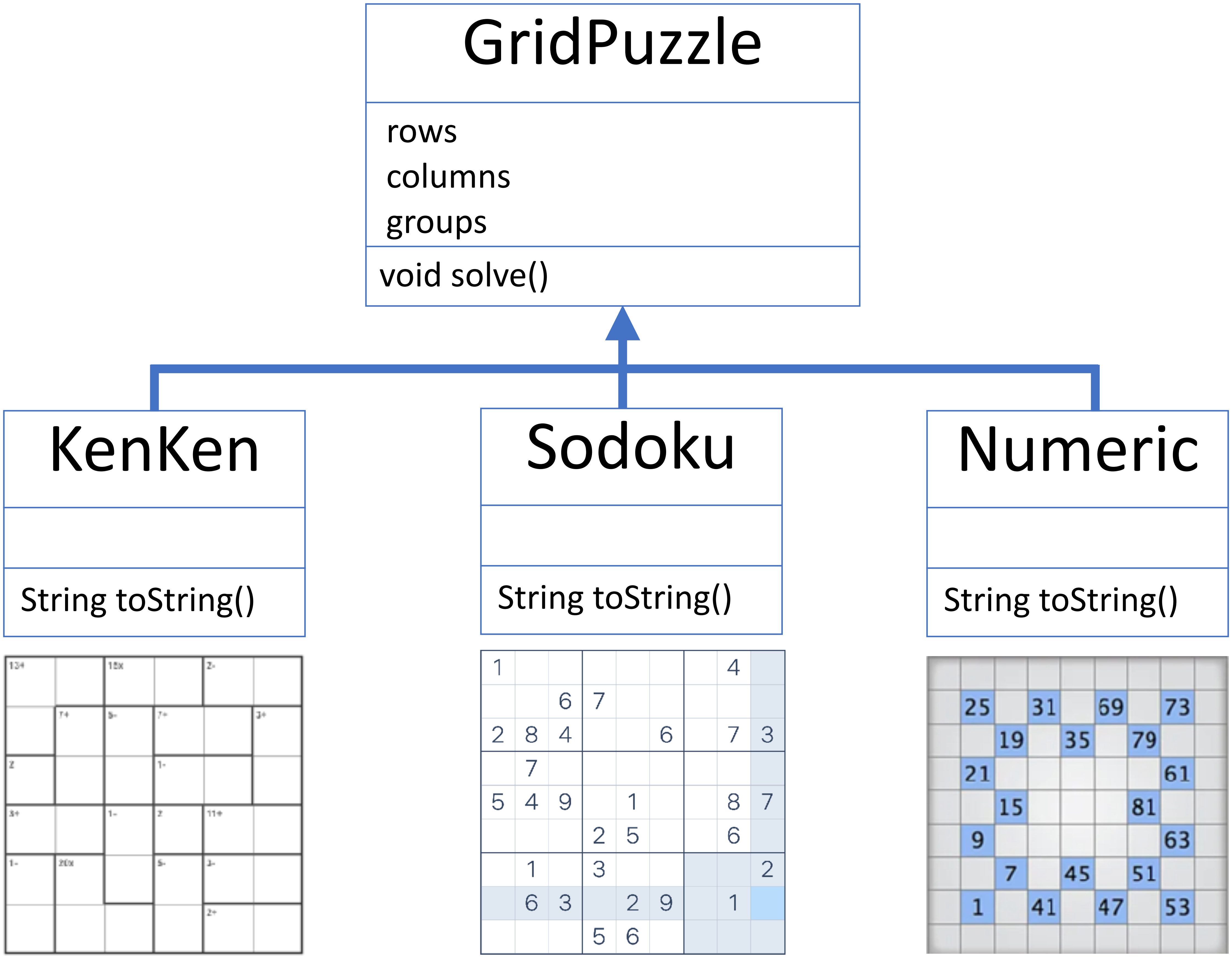
```
    while True:
        self.search_and_reduce_exclusive_cells()
        if self.is_solved():
            return

        self.search_and_reduce_matchlets()
        if self.is_solved():
            return
```

Code Reuse with Inheritance

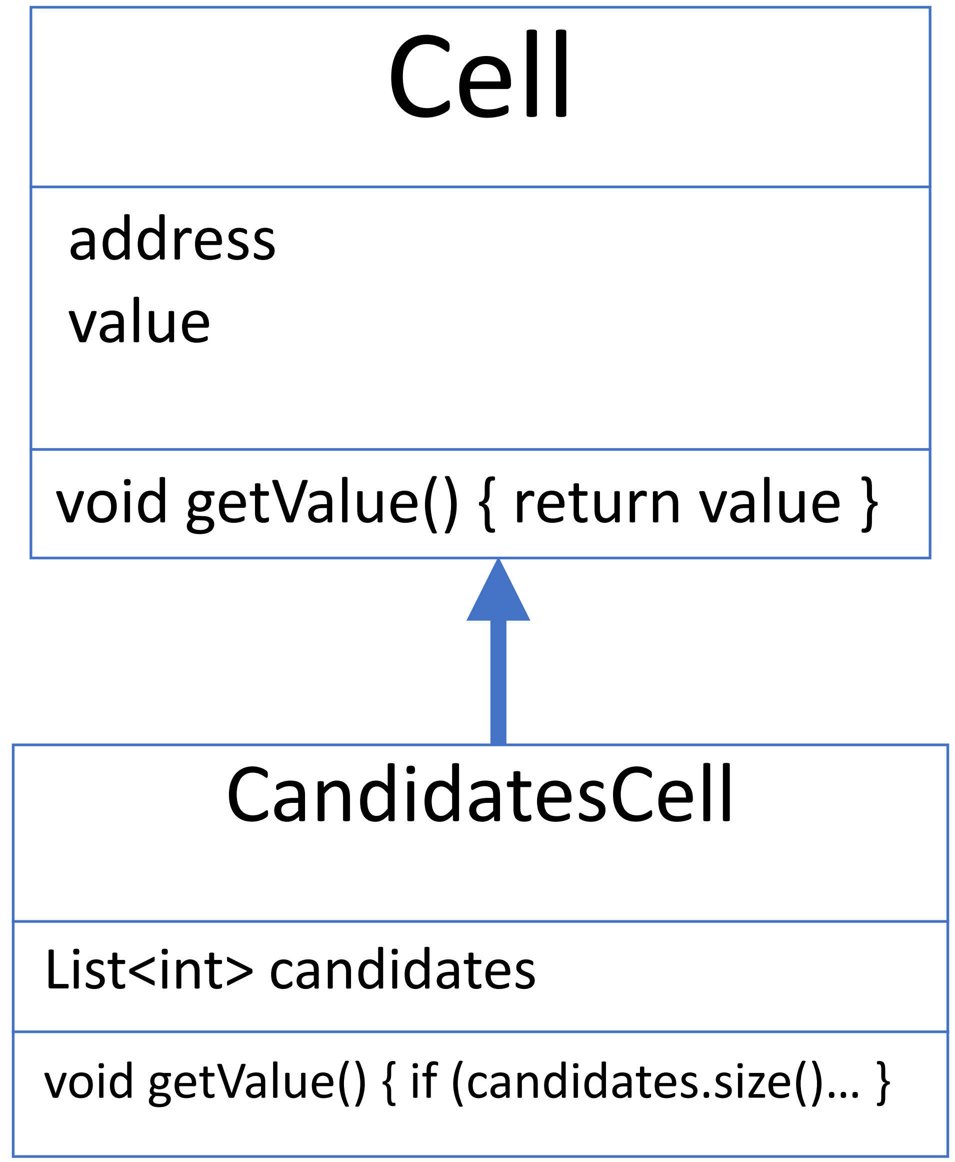
```
class Sudoku(GridPuzzle):

    def __init__(self, puzzle_string):
        super().__init__(puzzle_string)
        self.box_group_size = int(sqrt(self.size))
        self.column_boundaries = self.calculate_column_boundaries()
        self.row_boundaries = self.calculate_row_boundaries()
        self.box_groups = self.create_box_groups()
```

```
class GridPuzzle(object):

    def __init__(self, puzzle_definition, interactive=False):
        GridPuzzle.interactive_mode = interactive
        self.definition = puzzle_definition
        self.validate()
        self.size = self.calculate_size()

        self.column_names = [chr(ord('A') + col_number) for col_number in range(self.size)]
        self.row_names = [str(row_number + 1) for row_number in range(self.size)]

        self.given_cells = []
        self.guessed_cells = []

        self.puzzle_dict = self.create_puzzle()
        self.row_groups = self.create_row_groups()
        self.column_groups = self.create_column_groups()
```

Inheritance for Reuse



Inheritance for Reuse



Objects as Instances of a Class

- If you have two Student objects, **they each have their own data**

  e.g. Student A has a different set of values for its attributes than Student B
- But they both have the **same set of methods**
  - This is true because methods are associated with a **class** and the class acts as a template for creating new objects
  - We say "**Objects are instances of a class**"
- Classes define the complete behavior of their associated objects
  - what data elements and methods they have and how these features are accessed (whether they are public or private)

Classes, Subclasses, Superclasses

- The most important thing about a class is that it defines a type with a legal set of values
- Consider these four types
  - Complex Numbers ➡ Real Numbers ➡ Integers ➡ Natural Numbers
- Complex numbers is a class that includes all numbers; real numbers are a subtype of complex numbers and integers are a subtype of reals, etc.
  - in each case, moving to a subtype reduces the set of legal values
- The same thing is true for classes; A class defines a type and subclasses can be defined that excludes some of the values from the superclass

Class Inheritance

- Classes can exhibit **inheritance relationships**
  - Behaviors and data associated with a superclass are **passed down** to instances of a subclass
  - The subclass can **add new behaviors and new data** that are **specific to it**; it can also **alter behaviors that are inherited from the superclass** to take into account its own specific situation – making it a **derived class**
- It is extremely desirable that **any property that is true of a superclass is true of a subclass; the reverse is not true**: it is okay for properties that are true of a subclass not to be true of values in the superclass
  - For instance, the property isPositive() is true for all natural numbers but is certainly not true of all integers

Superclass/Subclass Inheritance Relationships

- Inheritance relationships are known as **is-a** relationships
  - Undergraduate **IS-A** Student (i.e. Undergraduate is-a subclass of Student)
- This phrase is meant to reinforce the concept that the subclass represents a more refined, more specific version of the superclass
- **If need be, we can treat the subclass as if it IS the superclass.**
  - It has all the same attributes and all the same methods as the superclass
- **so code that was built to process the superclass can equally apply to the subclass**

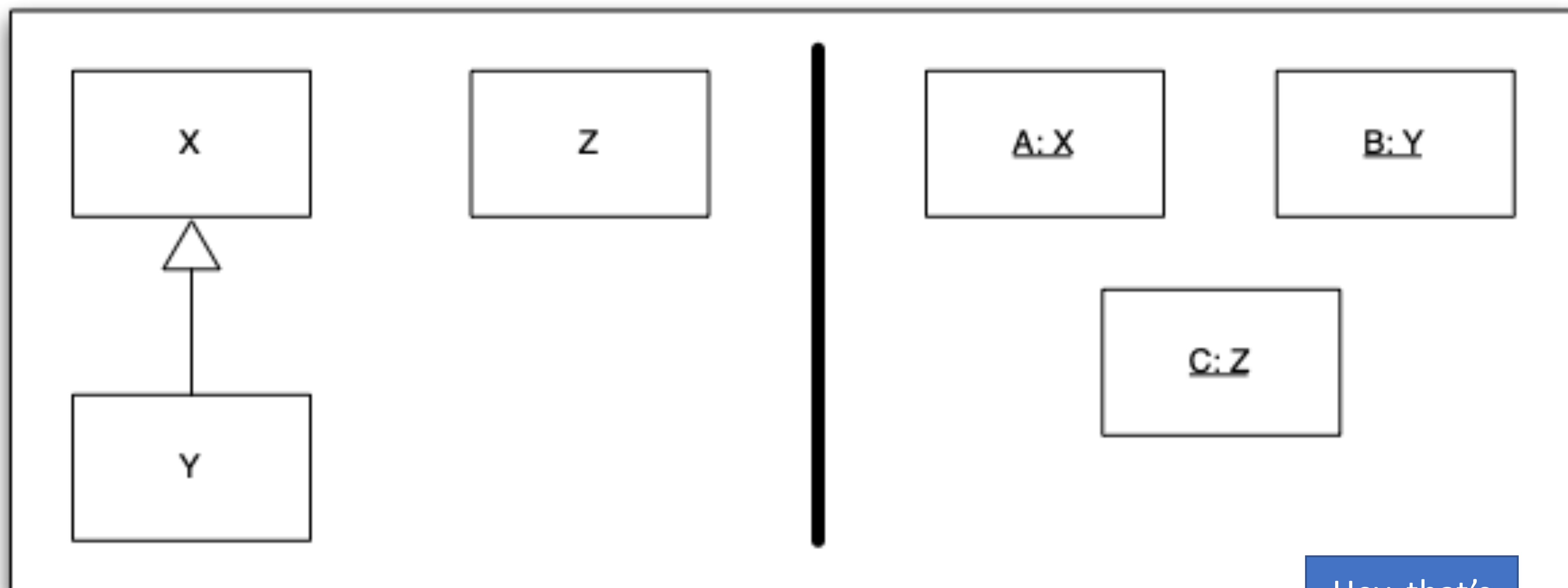
Liskov Substitution Principle

Class Encapsulation

- Classes can control the **accessibility of the features** of their objects
  - That is they can typically specify whether an attribute or method has an accessibility of public, protected, or private.
- This ability to **hide features of a class/module** is referred to as **encapsulation** or **information hiding**;
  - however, **encapsulation is a topic that is broader than just data hiding**, as we will discuss later in the semester

Accessibility, continued

- Object A is an instance of class X
- Object B is an instance of class Y which is a subclass of X;
- Object C is an instance of class Z which is unrelated to X and Y



Hey, that's a UML Model!

Accessibility, continued

- **Public** visibility of a feature of class X means that A, B and C can access that feature
- **Private** visibility of a feature of class X means that only A can access the feature
- **Protected** visibility of a feature of class X means that A and B can access the feature but C cannot.
- **Default** visibility: In Java the default accessibility is that any instance of a class defined in the same package as another class can access the instance variables and methods of an instance of that other class.

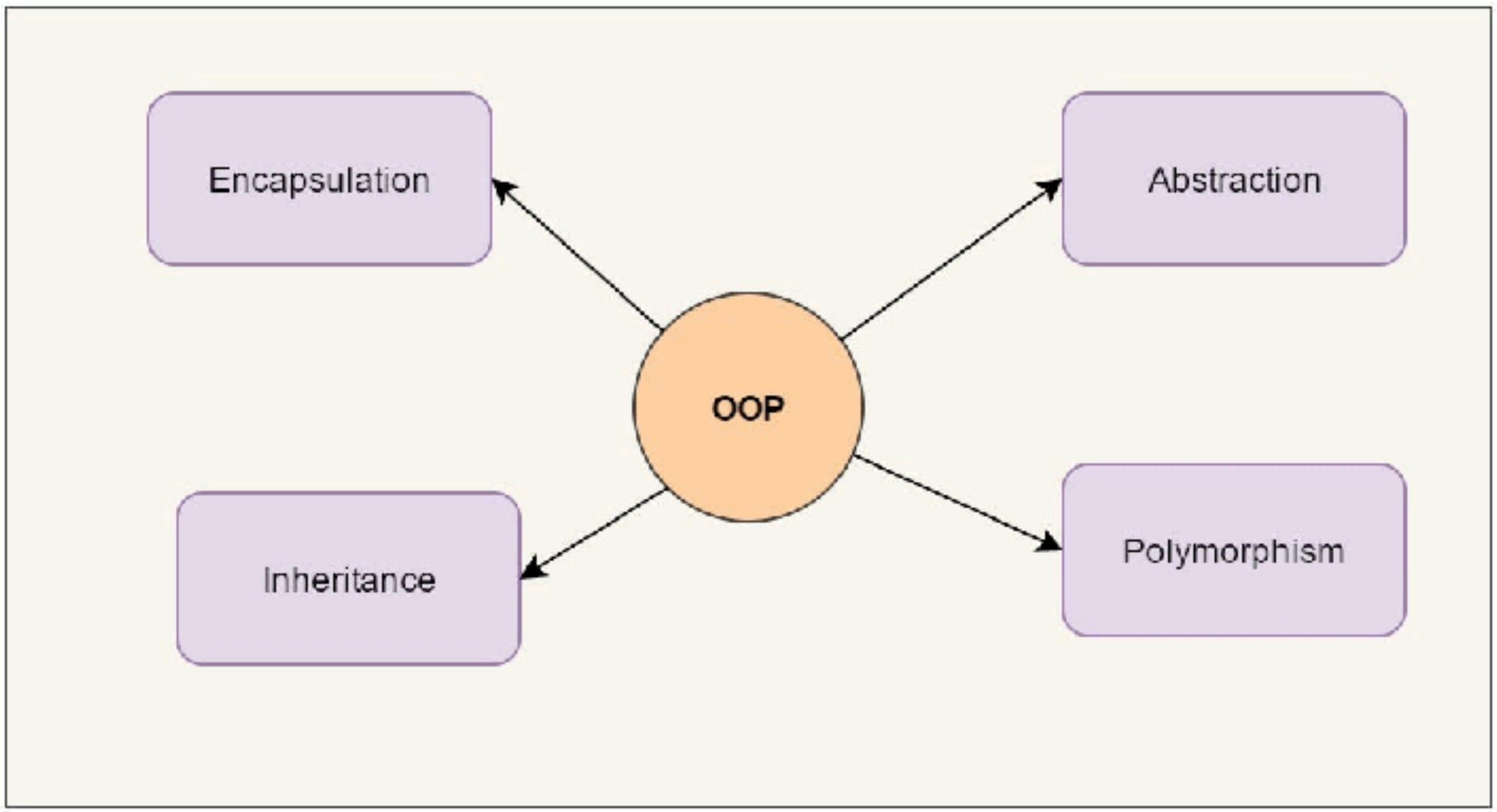
Class Constructors/Destructors

- OO Programming languages will (typically) provide “special methods” known as **constructors** and **destructors** to handle these two phases in an object’s life cycle
- Constructors are useful for ensuring that an object is properly initialized before any other object makes use of it
- Destructors are useful for ensuring that an object has released all of the resources it consumed while it was active.
- Java does NOT have a destructor in the same sense as C++. Java has the **finalize()** method, which ***might*** be called.

```java
try {
    System.out.println("Hi");
} catch (Exception ex) {
    // handle exception
} finally {
    // Clean up resources
}
```

Class Focus: OO Principles and Patterns

Principles First



Four Pillars of Object Oriented Programming

SOLID Principles

ingle Responsibility Principle — A class should have only one reason to change

pen-Closed Principle — Classes should be open for extension, but closed for modification

iskov Substitution Principle — Superclass objects should be replaceable by subclass objects without breaking functionality

nterface Segregation Principle — Clients should not have to implement methods in an interface they don’t use

ependency Inversion Principle — Depend on abstractions, not concrete classes

More OO Principles

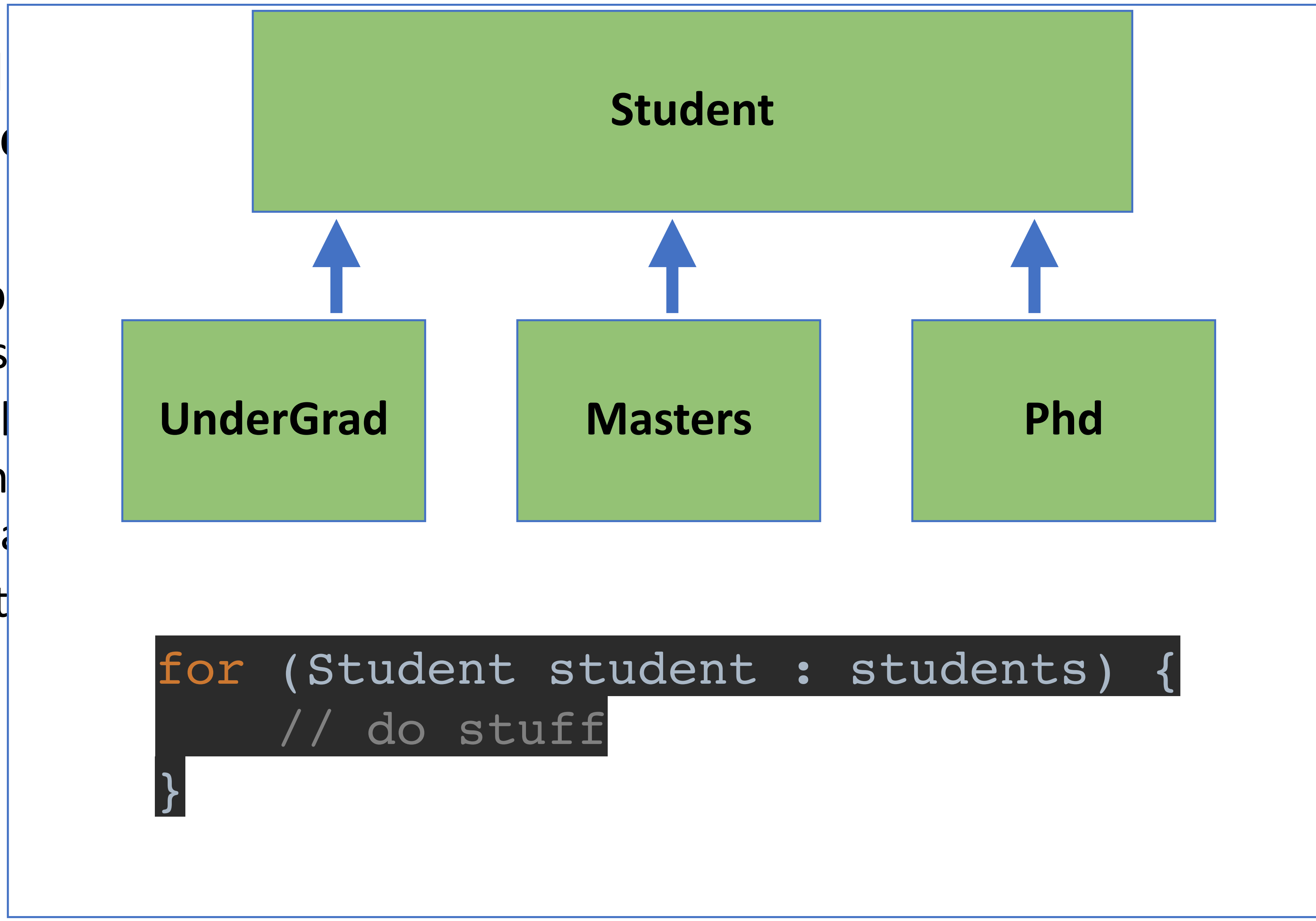
- Encapsulate what varies
- Favor composition (delegation) over inheritance
- Program to interfaces not implementations

Liskov Substitution Principle

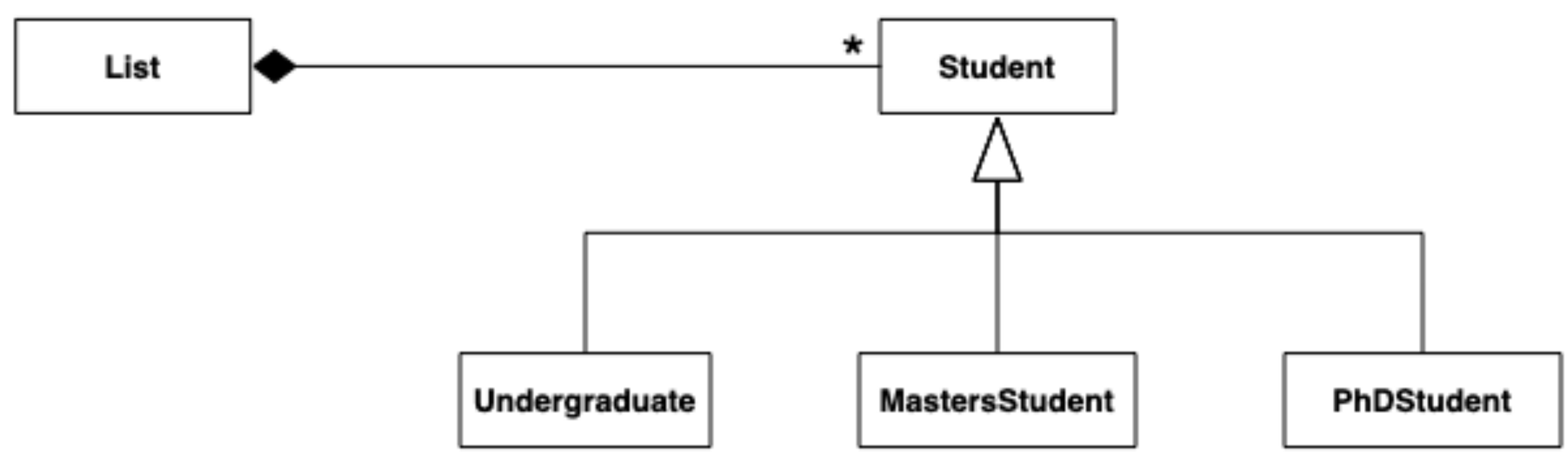
- Treat **al** **were all**
  instanc sses

Example:

- Supp DStudent
  in a s
- Probl e classes
  at on me and
  displ
- Solut t; You can
  then e same



Example rendered in UML



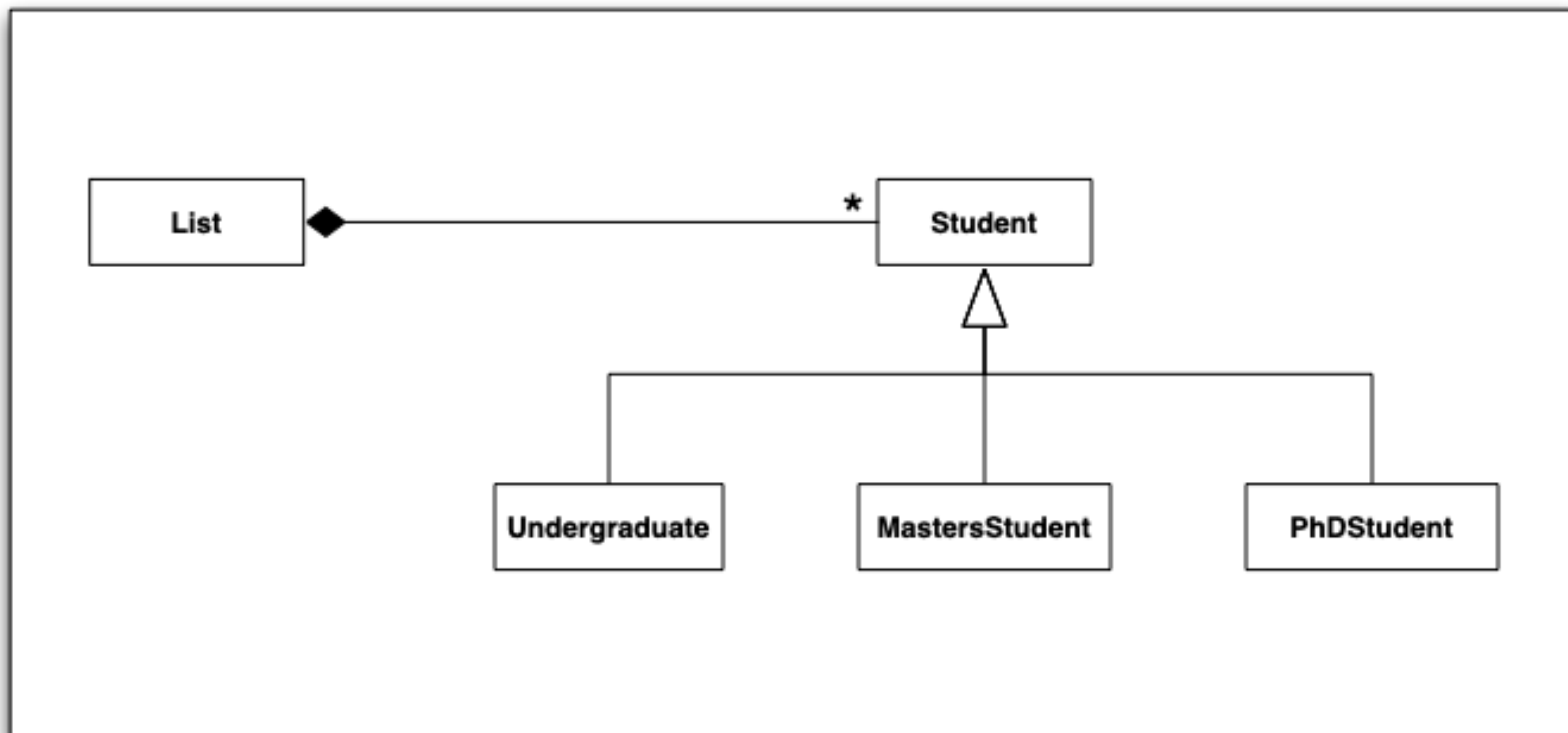
Note: UML Notation will be discussed in further lectures

Polymorphism

```java
import java.util.LinkedList;
import java.util.List;

public class Test {

    public static void main(String[] args) {

        List<Student> students = new LinkedList<Student>();

        students.add(new Undergraduate("Bilbo Baggins"));
        students.add(new MastersStudent("Aargorn"));
        students.add(new PhDStudent("Gandalf the White"));

        for (Student s: students) {
            System.out.println("" + s);
        }

        System.out.println();

        for (Student s: students) {
            s.saySomething();
        }

    }
```

Abstract Classes

- The classes that sit at the top of an object hierarchy are typically **abstract classes** while the classes that sit near the bottom of the hierarchy are called **concrete classes**



Summary

- In this lecture, we have touched on a variety of OO concepts
  - Functional Decomposition vs. the OO Paradigm
  - Requirements and Change in Software Development
  - Design perspectives: Conceptual, Specification, Implementation
- Be comfortable with the OO concepts and terminology
  - Coupling and Cohesion
  - Classes and Objects
  - Constructors, Destructors
  - Abstract, Derived, and Concrete Classes, Sub- and Super-class
  - Instance, Instantiation
  - Member, Attributes, Methods
  - Encapsulation
  - Inheritance
  - Polymorphism