

# **Introduction to Object-Oriented Programming**

# Outline

- Programming Paradigms
- Why Object-Orientation?
- Principles of Object-Oriented Programming
- Object-Oriented Programming Language Examples

# Programming Paradigms

- A *programming paradigm* is a way of conceptualizing what it means to perform computation, and how tasks that are to be carried out on a computer should be structured and organized.
- Paradigm: a way of seeing and doing things
- Programming paradigms are the result of people's ideas about how computer programs should be constructed
  - Patterns that serves as a “school of thoughts” for programming of computers
- Programming paradigms are a way to classify programming languages based on their features.
- Different paradigms provide different features and benefits
- Languages can be classified into multiple paradigms.
- Paradigms are **not meant to be mutually exclusive**
  - a single program can feature multiple paradigms!
- A lot of languages will facilitate programming in one or more paradigms.

# Programming Paradigms...

- One of the characteristics of a language is its support for particular programming paradigms.
- For example, Smalltalk has direct support for programming in the object-oriented way,
  - so it might be called an object-oriented language.
- OCaml, Lisp, Scheme, and JavaScript programs tend to make heavy use of passing functions around
  - so they are called “functional languages” despite having variables and many imperative constructs.

# Programming Paradigms...

- In Scala you can do imperative, object-oriented, and functional programming quite easily.
- If a language is *purposely* designed to allow programming in many paradigms is called a **multi-paradigm language**.
- Very few languages implement a paradigm 100%. When they do, they are **pure**. It is incredibly rare to have a “pure OOP” language or a “pure functional” language.

# Programming Paradigms...

- Programming Language
  - notation for specifying programs/computations
  - consists of words, symbols, and rules for writing a program
- Programming Paradigm
  - programming “technique”
  - way of thinking about programming
  - view of a program

# Programming Paradigms...

- The imperative paradigm includes:
  - the procedural paradigm
  - the structured paradigm
  - the object-oriented paradigm
- The declarative paradigm includes
  - the functional paradigm
  - the logical paradigm
- Some paradigms are more appropriate for one problem than another, allows you to select best approach
  - Object oriented great for database design
  - Imperative great for scientific programming

# Imperative Programming Paradigm

- This paradigm was one of the earliest and was developed using machine-language
- It is also the basis on which all hardware is implemented
- statements are instructions at the native machine-level, and they contain states and variables that point right to the memory
- close to the machine
  - fast execution time
  - more efficient
- its weaknesses:
  - order sensitive
  - the limitation of abstraction



# Imperative Programming Paradigm...

- ```
int fact(int n) {  
    int f = 1;  
    while (n > 1) {  
        f = f * n;  
        n = n - 1;  
    }  
    return f;  
}
```
- \* Notice the assignment operator, the equals (=) sign.  
The value location of the variable itself was stored in memory

# Imperative Programming Paradigm...

- ```
result = []  
i = 0  
start:  
    numPeople = length(people)  
    if i >= numPeople goto finished  
    p = people[i]  
    nameLength = length(p.name)  
    if nameLength <= 5 goto nextOne  
    upperName = toUpper(p.name)  
    addToList(result, upperName)  
nextOne:  
    i = i + 1  
    goto start  
finished:  
    return sort(result)
```

# Imperative Programming Paradigm...

- It is a programming paradigm that uses statements that change a program's state
  - Control flow in **imperative programming** is *explicit*
    - commands show *how* the computation takes place, step by step.
  - Each step affects the global **state** of the computation.
- An imperative program consists of commands for the computer to perform.
- Imperative programs describe the details of **HOW** the results are to be obtained
- HOW means describing the inputs and describing how the outputs are produced
- Example: C, C++, Java, PHP, Python, Ruby

# Procedural Paradigm

- It describes, step by step, exactly the procedure that should, according to the particular programmer at least, be followed to solve a specific problem.

# Structured Paradigm

- **Structured programming** is a kind of imperative programming where control flow is defined by nested loops, conditionals, and subroutines, rather than via gotos.
- Variables are generally local to blocks (have lexical scope).
- ```
result = [];  
for i = 0; i < length(people); i++ {  
    p = people[i];  
    if length(p.name) > 5 {  
        addToList(result, toUpper(p.name));  
    }  
}  
return sort(result);
```

## Structured Paradigm...

- Early languages emphasizing structured programming include:
  - Algol 60, PL/I
  - Algol 68
  - Pascal
  - C
  - Ada 83
  - Modula
  - Modula-2

# OOP Paradigm

- It borrows from all the major paradigms
- It is based on the concepts of “objects”, which may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods
- There is significant diversity of OOP languages, but the most popular ones are class-based, meaning that objects are instances of classes, which typically also determine their type
- In OOP, computer programs are designed by making them out of objects
- Examples: C++, C#, Java, PHP, Python

# OOP Paradigm...

- **public class Example{  
private String name;  
public void setName(String n){name = n;}}**
- Class Example has a method *setname* and a string *n* that is assigned to a variable *name* in memory.



# Declarative Programming Paradigm

- It is a paradigm that expresses the logic of a computation without describing its control flow
- Control flow in **declarative programming** is *implicit*
  - the programmer states only *what* the result should look like, **not** how to obtain it.
- It focuses on what the program should accomplish
- It often considers programs as theories of a formal logic, and computations as deductions in that logic space
- Examples: SQL, XSQL (XMLSQL)

# Declarative Programming Paradigm

- `select upper(name)`  
`from people`  
`where length(name) > 5`  
`order by name`
- No loops, no assignments, etc. Whatever engine that interprets this code is just supposed to get the desired information, and can use whatever approach it wants. (The logic and constraint paradigms are generally declarative as well.)

# Functional Programming Paradigm

- It is a subset of declarative programming
- This paradigm is thought of to have been originated from a mathematical discipline
- Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions
- control flow is expressed by combining function calls, rather than by assigning values to variables
  - It discourages changes in the value of variables through assignment, making a great deal of use of recursion instead
- is essentially less complex and offers better readability than imperative.

# Functional Programming Paradigm...

- There are no commands, only side-effect free expressions
- Code is much shorter, less error-prone, and much easier to prove correct
- There is more inherent parallelism, so good compilers can produce faster code

# Functional Programming Paradigm...

- Merits include:
  - compare to imperative it has a higher level of abstraction, is not tied to dependency.
- Some of the weakness include:
  - less efficiency
  - troubleshooting variables or it's sequential activities are better handled in both Object-Oriented or imperatively
- Example:  
**Function FindAvg(int x, int y){ return ((x+y)/2);}**
- Programming Languages in this paradigm: F#, Lisp, Python, Ruby, JavaScript

# Functional Programming Paradigm...

- `sort(  
 fix( $\lambda$ f.  $\lambda$ p.  
 if(equals(p, emptylist),  
 emptylist,  
 if(greater(length(name(head(p))), 5),  
 append(to_upper(name(head(p))), f(tail(p))),  
 f(tail(people)))))(people))`

# Logical Paradigm

- It focuses primarily on predicate logic — relation.
- This is also a vital part of the logic circuit of a computer.
- Merits of this paradigm include:
  - problems are solved by the system and Proving the validity of a given program is simple

# Logical and Constraint Paradigm...

- **Logic programming** and **constraint programming** are two paradigms in which programs are built by setting up relations that specify **facts** and inference **rules**, and asking whether or not something is true (i.e. specifying a **goal**.)
- Unification and backtracking to find solutions (i.e.. satisfy goals) takes place automatically.



# Logical Paradigm...

- **Brother(X,Y)**      **/\* X is the brother of Y \*/**  
    **:-**      **/\* if there are two people F and M for which \*/**  
    **father(F,X),**      **/\* F is the father of X \*/**  
    **father(F,Y),**      **/\* and F is the father of Y \*/**  
    **mother(M,X),**      **/\* and M is the mother of X \*/**  
    **mother(M,Y),**      **/\* and M is the mother of Y \*/**  
    **male(X).**      **/\* and X is male \*/**
- Interpretation: X is the brother of Y if they have the same father and mother and X is male
- Languages that emphasize this paradigm:
  - **Prolog**
  - **GHC**
  - **Parlog**
  - **Vulcan**
  - **Polka**
  - **Mercury**
  - **Fnil.**

# Mixed

- ```
/**
 * This RobotTester java class is created to run and execute the Robot
 * java class, which is designed to direct the robot based on a users input and
 * calculate the position of the robot in comparison to a starting coordinate.
 * author@Geo,Wade
 */
import java.util.Scanner;

public class RobotTester
{
    public static void main(String []args)
    {
        // part 1: open a scanner
        Scanner scan = new Scanner(System.in);

        // part 2: Abstract the Robot class
        Robot rTwoD = new Robot();

        // part 3: prompt the user to enter a direction and move the robot
        boolean finished = true;
```

# Mixed...

- ```
while (!finished)
{
    System.out.println("Enter 1 to turn Right,
Enter 2 to turn Left, Enter 3 to move, Enter Q to
quit ");
    int turn = scan.nextInt();

    if (turn == 1)
    {
        rTwoD.turnRight(turn);
    }
}
```

# Mixed...

```
else if (turn == 2)
{
    rTwoD.turnLeft(turn);
}
else if (turn == 3)
{
    String facing = rTwoD.getDirection();
    System.out.println("The Robot is facing: " + facing);
}
else finished = false;

}

// part 4: close the scanner
scan.close();
System.exit(0);

}

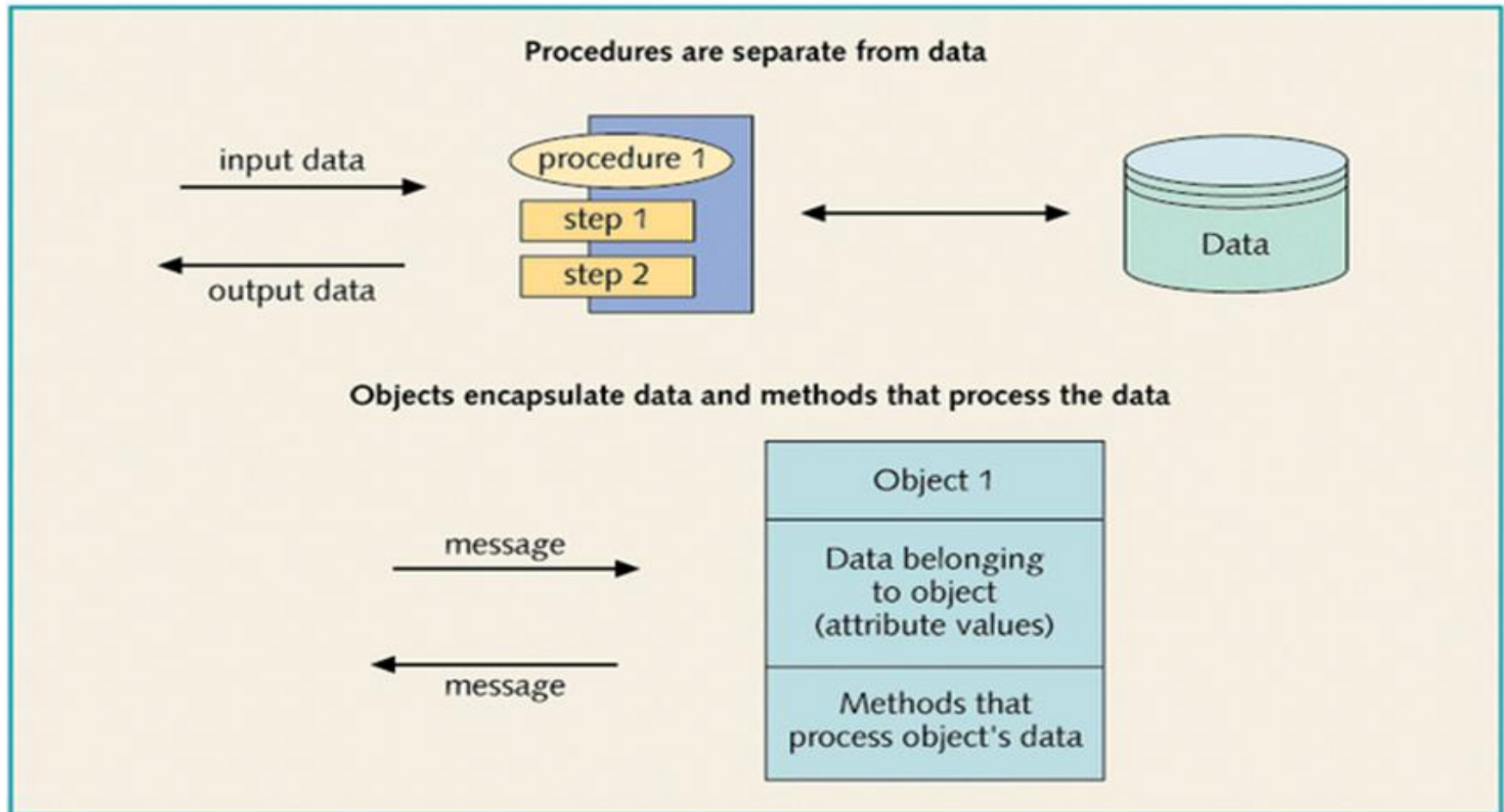
}
```

# Why Object-Orientation?

# Procedural vs. OO Approaches

- Procedural approach
  - System is defined as a set of procedures that interact with data
    - Data is maintained separately from procedures
- OO approach
  - System is defined as a collection of objects that work together to accomplish tasks
    - Objects carry out actions when asked
    - Each object maintains its own data

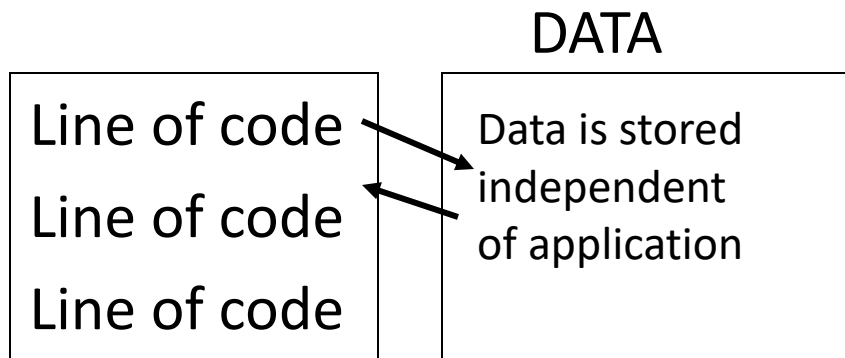
# Procedural vs. OO Approaches...



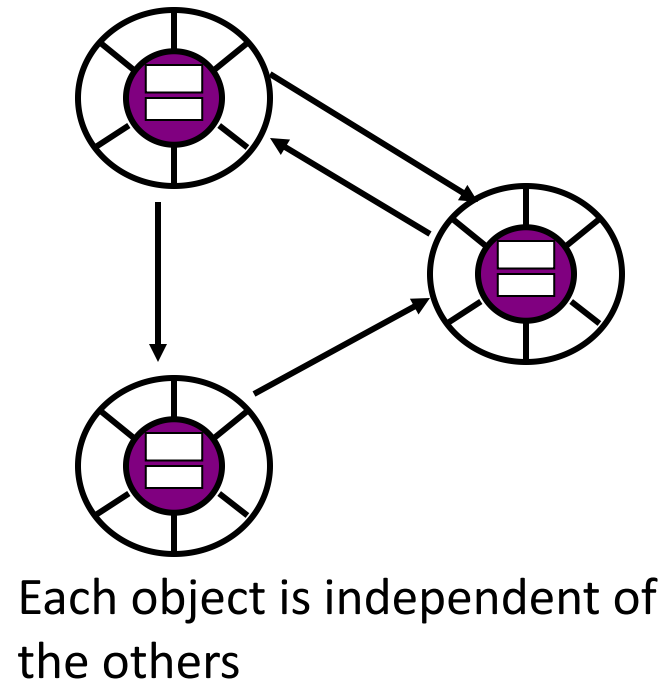
Procedural approach versus object-oriented approach

# Procedural vs. OO Approaches...

## Procedural application



## OO-application





# Advantages of OOP Sys dev't

- Objects are more natural
- Reuse
  - Classes and objects can be invented once and used many times during analysis, design, and programming
  - Do not need source code for reused class, simply need to know interface

# Principles of OOP

- Abstraction
- Encapsulation
- Inheritance
- polymorphism

# Abstraction

- It refers to the process of hiding the details and exposing only the essential features of a particular concept or object
- The process of taking away or removing characteristics from something in order to reduce it to a set of essential characteristics

# Abstraction...

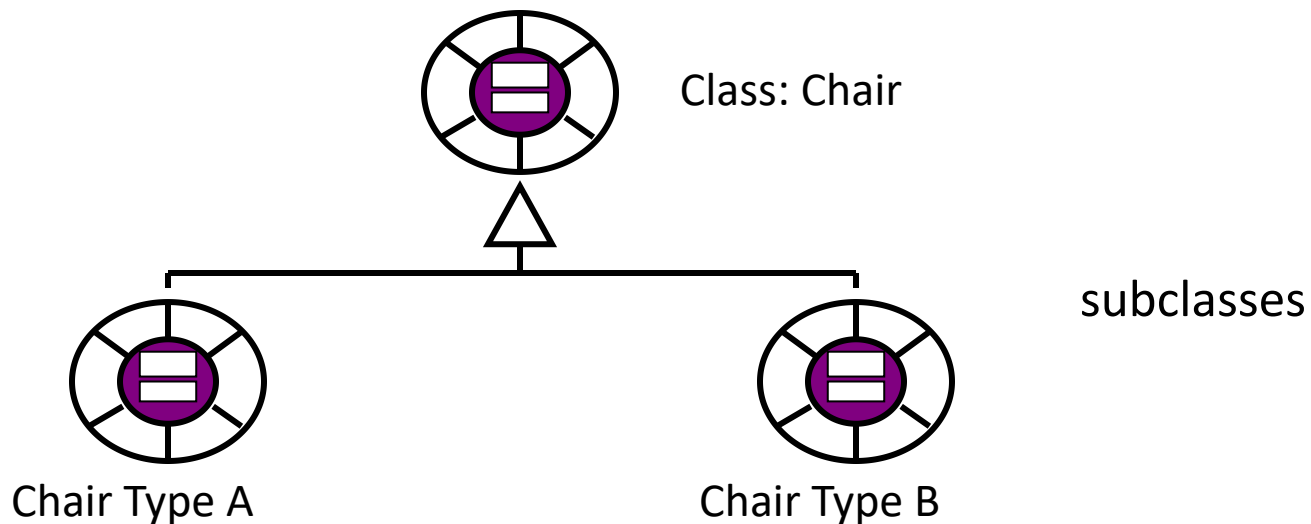
- Binding the data with the code that manipulates it.
- It keeps the data and the code safe from external interference
- Separation between concepts and details.
- Objects and classes provide abstraction in programming.

# Encapsulation

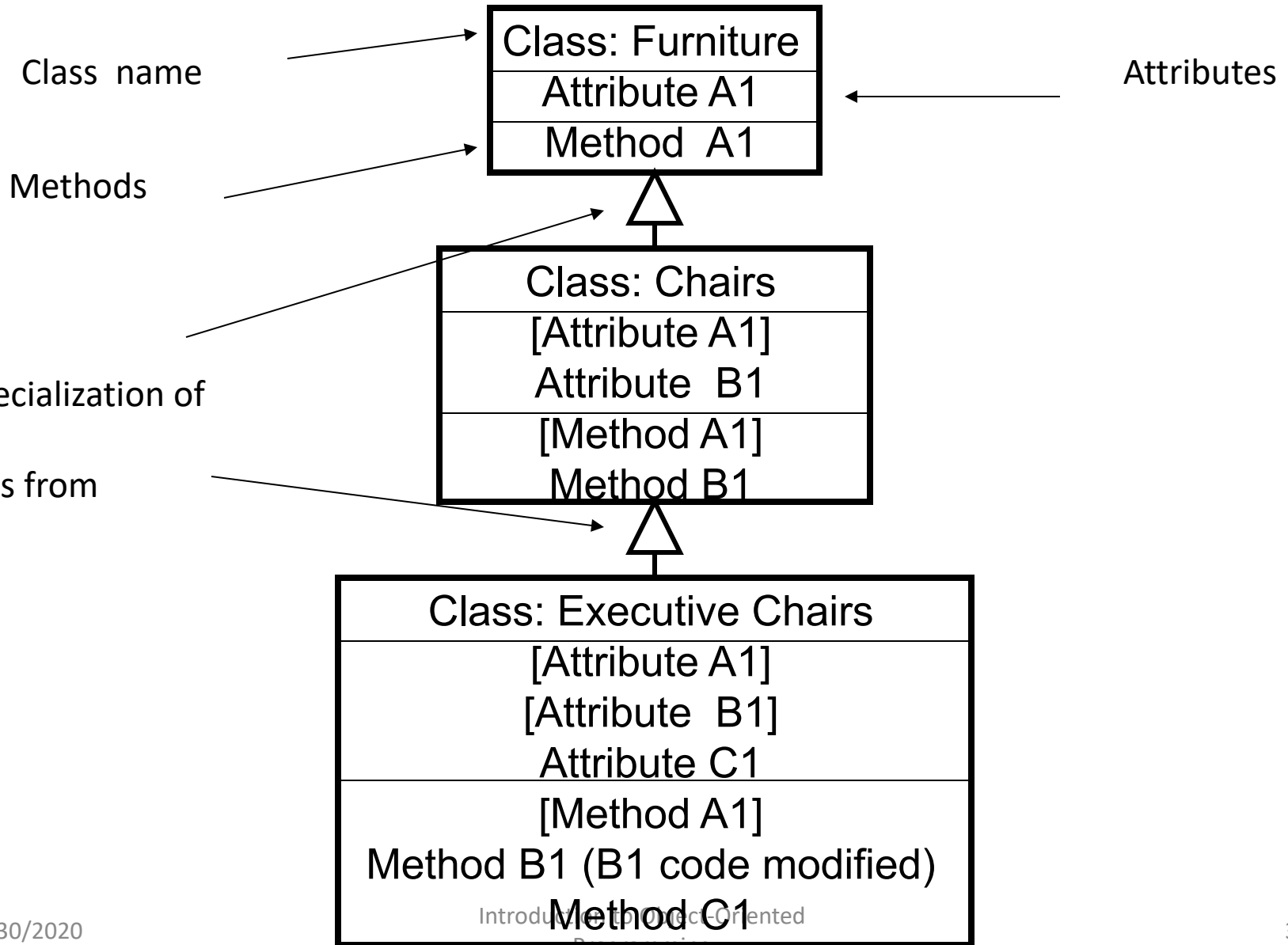
- Each objects methods manage it's own attributes.
- This is also known as *hiding*.
- An object A can learn about the values of attributes of another object B, only by invoking the corresponding method (message) associated to the object B.
- Example:
  - Class: Lady
  - Attributes: Age, salary
  - Methods: get\_age, set\_salary

# Inheritance

- Inheritance is the mechanism by which an object acquires the some/all properties of another object.
- It supports the concept of hierarchical classification.
- Classes can be arranged in hierarchies so that more classes inherit attributes and methods from more abstract classes



# Class Inheritance & Specialization



# Polymorphism

- Polymorphism means to process objects differently based on their data type.
  - one method with multiple implementation, for a certain class of action
  - which implementation to be used is decided at runtime depending upon the situation (i.e., data type of the object)
- This can be implemented by designing a generic interface, which provides generic methods for a certain class of action
  - there can be multiple classes, which provides the implementation of these generic methods.



# Polymorphism...

- Polymorphism could be static and dynamic both.
  - **Method Overloading** is static polymorphism while,
  - **Method overriding** is dynamic polymorphism.
- **Overloading** in simple words means more than one method having the same method name that behaves differently based on the arguments passed while calling the method.
- This called static because, which method to be invoked is decided at the time of compilation
- **Overriding** means a derived class is implementing a method of its super class.
- The call to overridden method is resolved at runtime, thus called runtime polymorphism

# OOP Languages

- SmallTalk
- C++
- J++
- C#
- Java