Basic Java
Unit 3 – OO in Java

Pratian Technologies (India) Pvt. Ltd.
www.pratian.com

PRATIAN
TECHNOLOGIES

Topics

☞ What is OOP?

☞ OOP v/s Structured Programming

☞ Classes & Objects

☞ Access Specifiers

☞ Constructors

☞ Static Members

☞ Encapsulation

☞ Abstraction

☞ Inheritance

☞ Polymorphism

☞ Final modifier

☞ Inner Classes

Basic Java
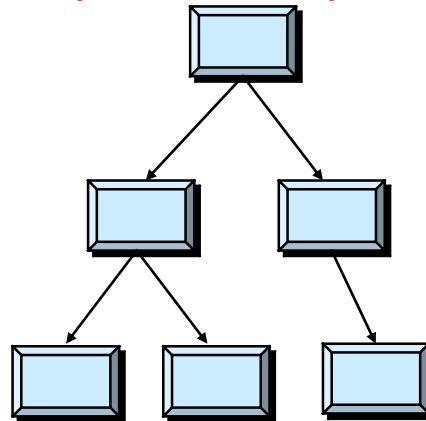
The procedural approach

- The procedural approach
  - Deals with functions as the **building blocks**
  - Easy to start with
  - Higher comfort level for a new programmer
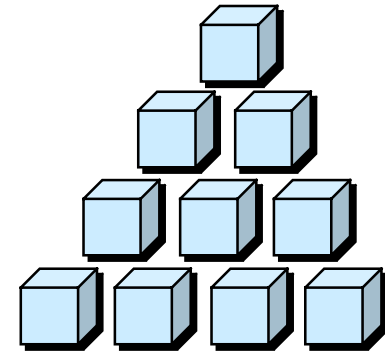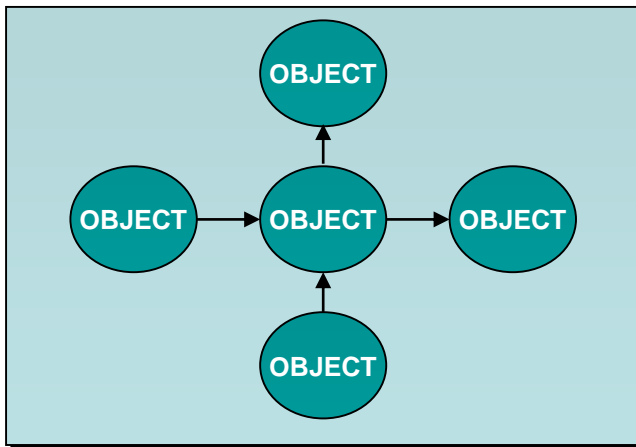
Functions are the building blocks

- Simple decomposition technique for
  - Modularity
  - Reusability
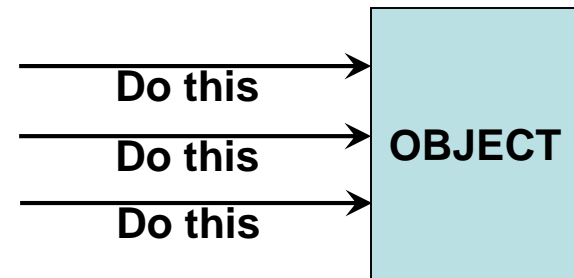 - Complexity

Top – down decomposition

OO Approach

- An Object oriented approach views systems and programs as a collection of interacting objects.

- An object is a thing in a computer system that is capable of responding to messages

What are Objects?
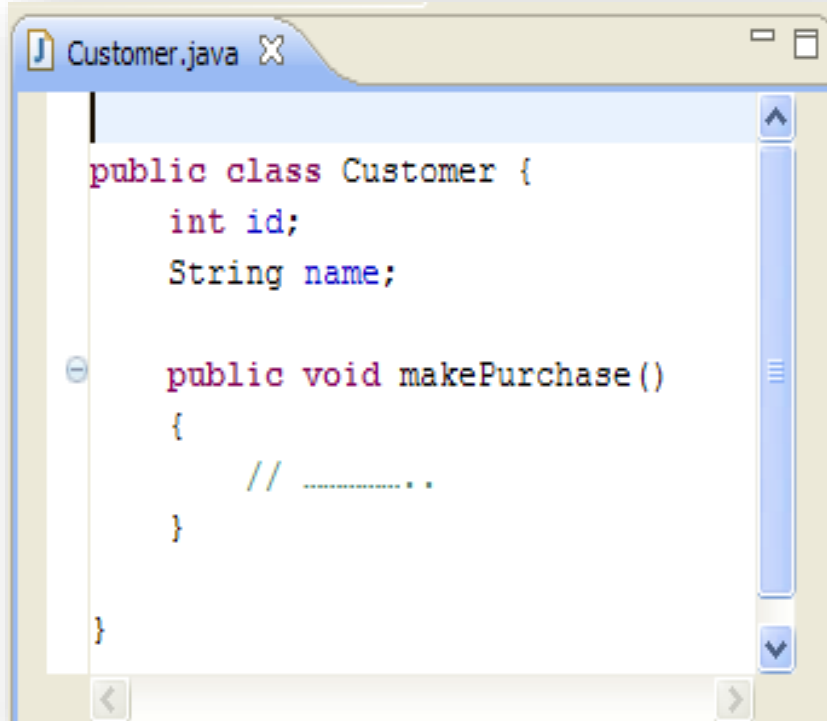
- ## We interact with *objects* everyday
  - ### A customer
  - ### An order
  - ### Your car
  - ### The telephone

- ## An object represents an entity – physical, conceptual or software

  - ### **Physical entity**
    - #### Employee, Customer, Supplier

  - ### **Conceptual entity**
    - #### Sales, Policy, TaxCalculator

  - ### **Software entity**
    - #### Linked List, Connection, etc.

  - ### *A programmer should make a good effort to capture the conceptual entities in addition to physical entities which are relatively straight forward to identify*

Why choose the OO approach?

PRATIAN
TECHNOLOGIES
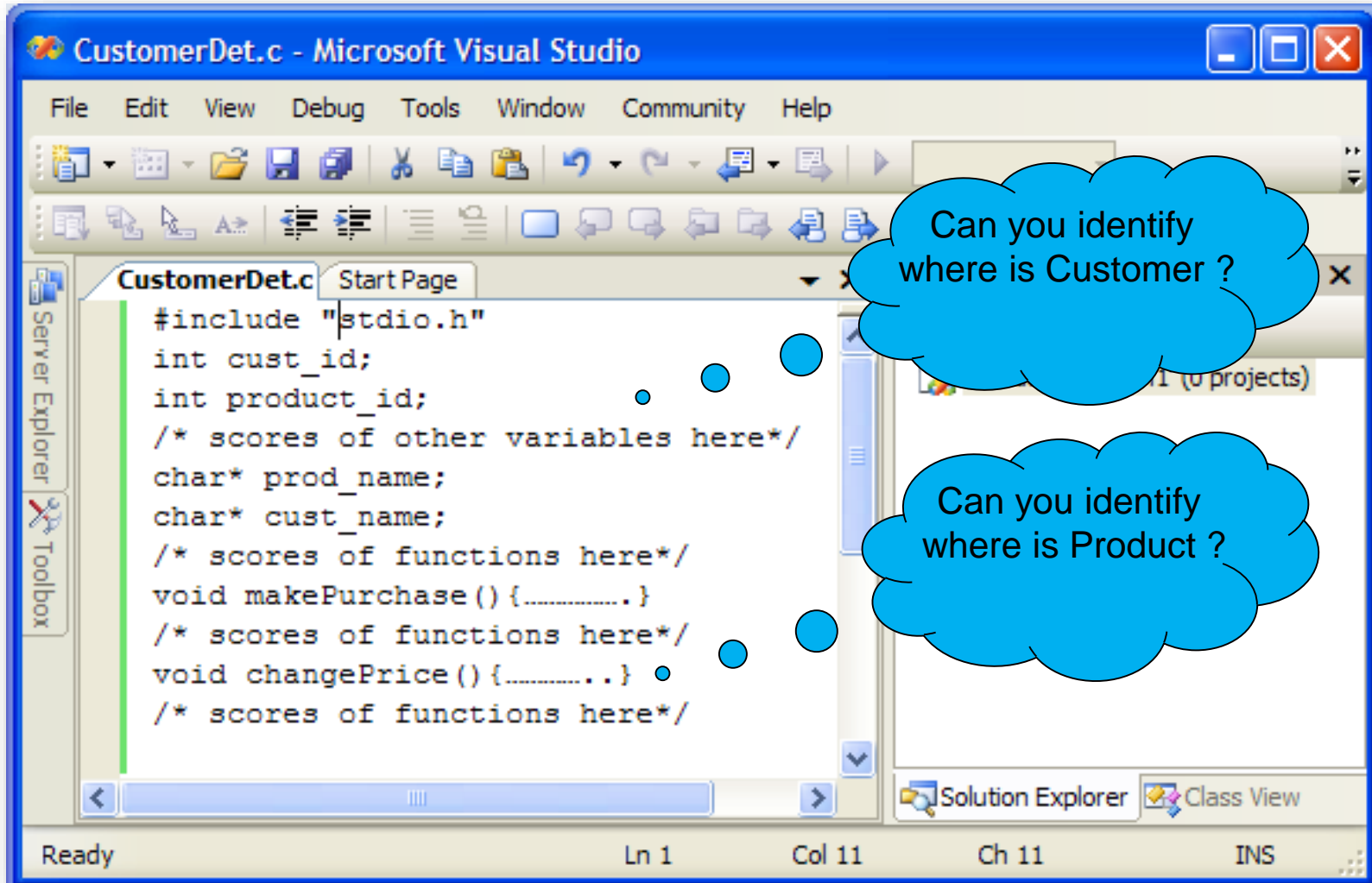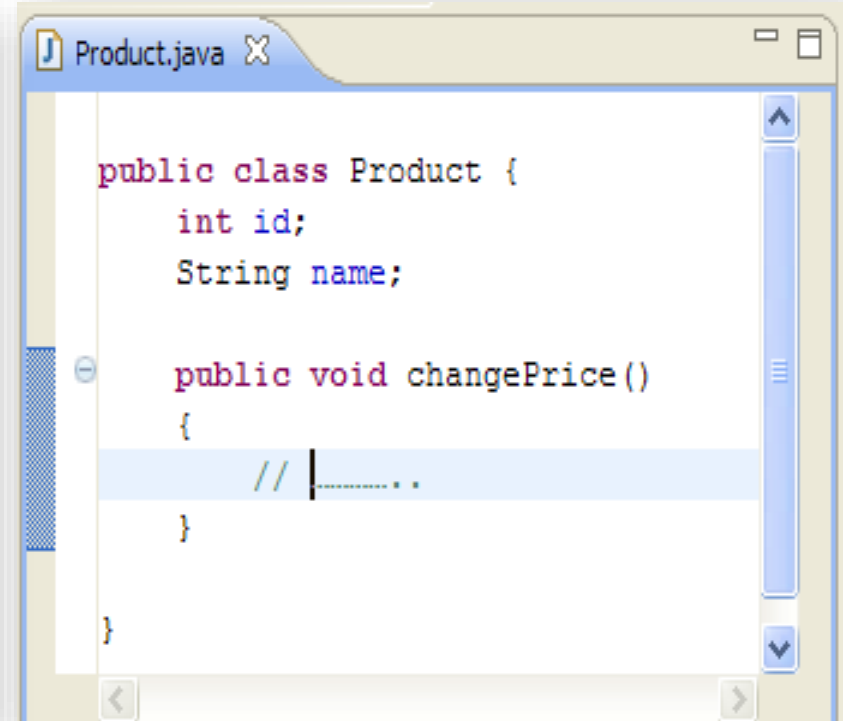
Why choose the OO approach ?

```java
Customer.java

public class Customer {
    int id;
    String name;

    public void makePurchase()
    {
        // ...............
    }

}
```

```java
Product.java

public class Product {
    int id;
    String name;

    public void changePrice()
    {
        // |...........
    }

}
```
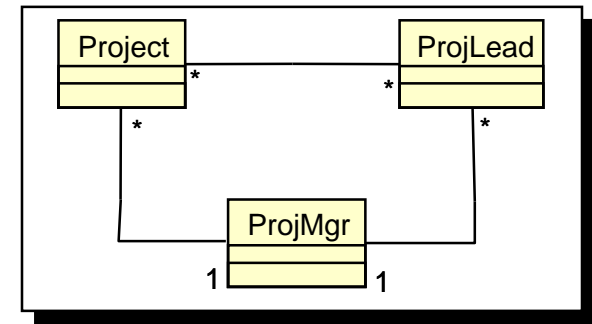
Easier to comprehend

Basic Java

Why choose the OO approach?

- The OO approach
  - Deals with classes as the building blocks
  - Allows Real World Modeling
  - The idea of OOP is to try to approach programming in a more natural way by grouping all the code that belongs to a particular object—such as an account or a customer — together

- Raise the level of abstraction
  - Applications can be implemented in the same terms in which they are described by users

- Easier to find nouns and construct a system centered around the nouns than actions in isolation
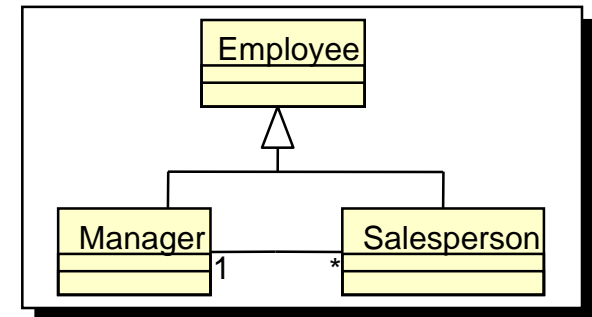
- Easier to visualize an encapsulated representation of data and responsibilities of entities present in the domain

- The modern methodologies recommend the object-oriented approach even for applications developed in C or Cobol

Object-Oriented Programming

"Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class..."   **Grady Booch**

Procedural vs. Object-Oriented Programming

- The unit in procedural programming is *function*, and unit in object-oriented programming is *class*

- Procedural programming concentrates on creating functions, while object-oriented programming starts from isolating the classes, and then look for the methods inside them.

- Procedural programming separates the data of the program from the operations that manipulate the data, while object-oriented programming focus on both of them
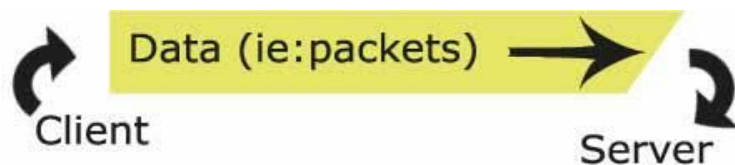
figure1: procedural
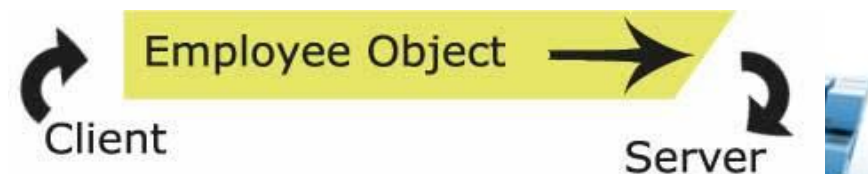
figure2: object-oriented

Basic Java

What is a Class?

Total surface area = 144 m² (122 m²)

STUDIO

TOILET

LUMBER ROOM

BOILER

HALLWAY 2

KITCHEN

LIBRARY

HALLWAY 1

DINING ROOM

LIVING ROOM

Basic Java
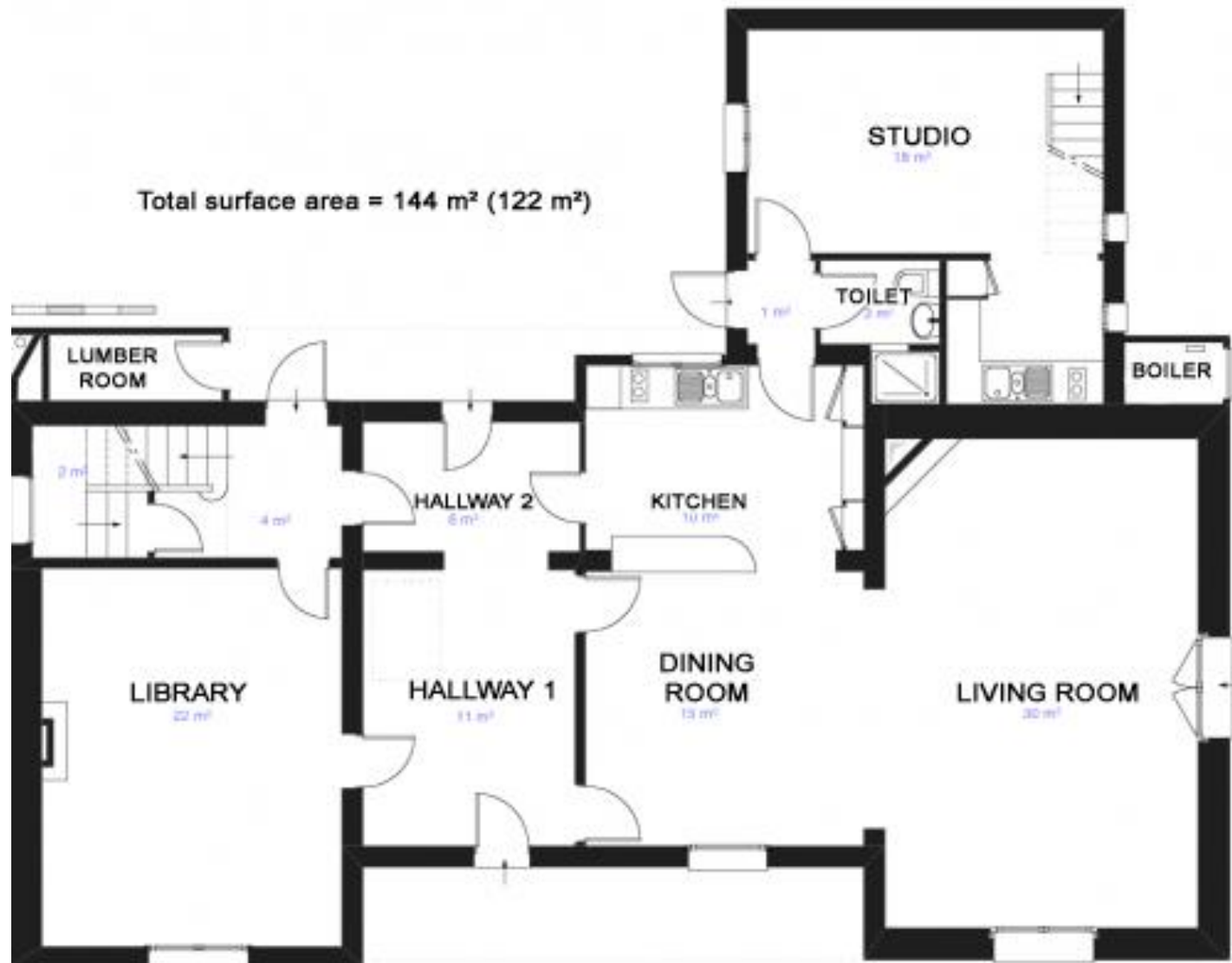
What is a Class?

What is a Class?

A **class** gives a generic notion of what objects look like

An employee
Joe

Employee
Class

Instantiate (create)

An **object** is an instance of a class

Classes are used to distinguish one type of object from another

Employee

Reg. Customer

Customer
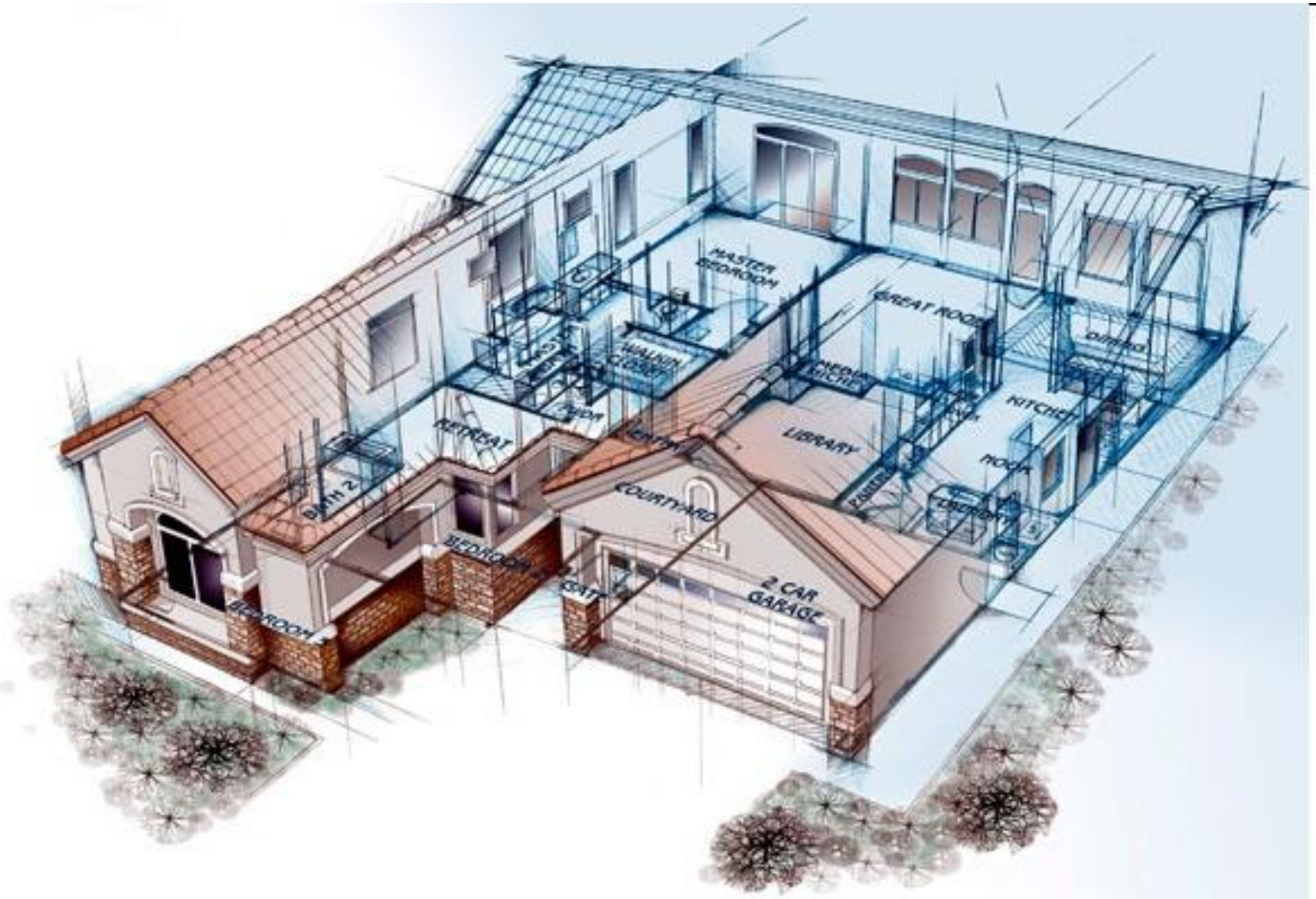
Types
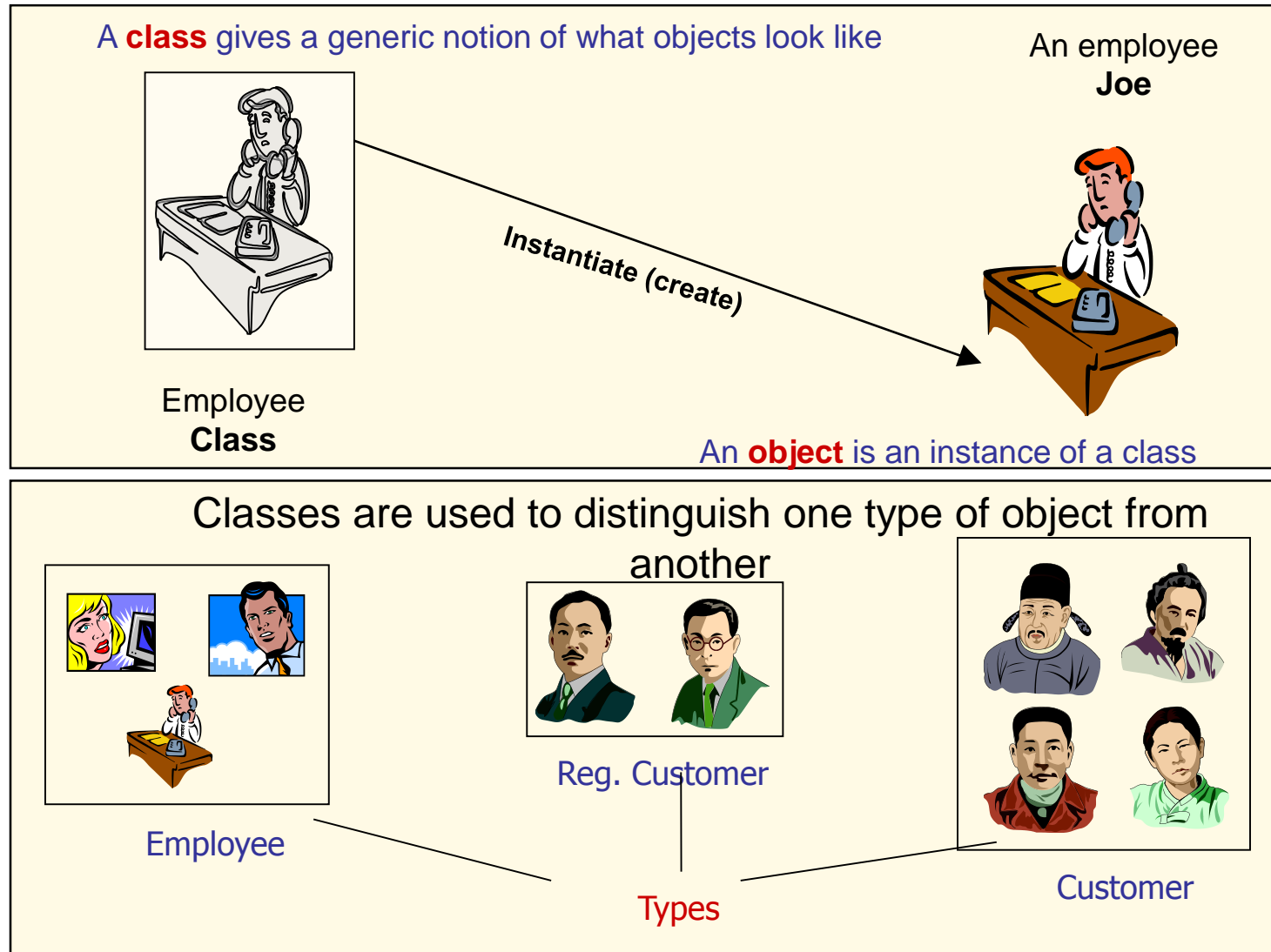
Class

- ## User defined type

  - Encapsulates all the data and operations pertaining to an entity

  - Provides a Single representation to all the attributes defining the entity

  - Passing single representations is easier

Employee
🔒 empId : String
🔒 name : String
🔒 address : Address
◆ getEmpID() : String
◆ setEmpId(empId : String)
◆ getName() : String
◆ setName(name : String)
◆ getAddress() : Address
◆ setAddress(address : Address)

- ## Data types as collections

  - A struct in C encapsulates only data. Used as a data structure to store different types of data

  - An array is used to store different elements of the same type

Relationship between Classes

- Classification

<<Is-a>>	**<<Has-a>>**	**<<Uses>>**
Generalization Realization	Association Aggregation Composition	Dependency

- For all practical purposes we will represent

  - Is-a relationship as

    ⟶
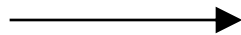
  - Has-a relationship as

    ⎯⎯⎯⎯

  - Uses relationship as

    - - - - - - - - - ->
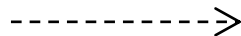
Has-a Relationship

- The 'Has-a' relationships are qualified by
  - Multiplicity
    - The number of instances with which a class is associated
    - Can be 1, 0..1, *, 1..*, 0..*, 2..*, 5..10, etc.
    - Multiplicity is by default 1
  - Navigability
    - Can be unidirectional or bidirectional
    - Navigability is by default bi-directional
  - Role name
    - The name of the instance in the relationship
    - Multiple 'has-a' based on different roles are possible

Employee

-curr 1

-prev

0..*

Employment
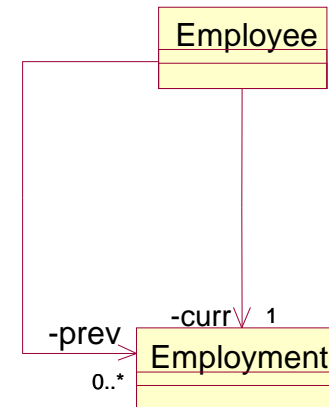
Identifying Classes

> A trainer trains many trainees on a given technology in this course, which contains many modules – each module is comprised of different units and each unit has many topics.

- Identify the different classes from the above problem statement

Procedural approach

- Focus is on identifying **VERBS**
- Connections between functions established through Function Calls

OO approach

- Focus is on identifying **NOUNS**
- Connections between classes established through Relationships ('Is-a' and 'Has-a')

Identifying Classes

- Trainer

- Trainee

- Course

- Technology

- Module

- Unit

- Topic

- Identify the different connections (relationships) between the above classes

Identifying Relationships

- Trainer - Trainee
    - Trainer 'HAS' many Trainees
    - Every Trainee 'HAS' a Trainer

Trainer	*has*	Trainee
1	*	
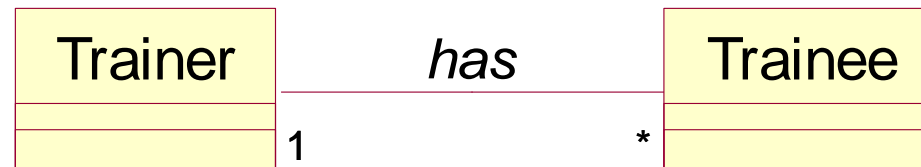
Identifying Relationships

- Course – Technology
- Course - Module
  - Course 'HAS' an associated Technology
  - A Technology has many courses
  - Course 'HAS' many Modules

Course		

* *has* 1 **Technology**

has

*
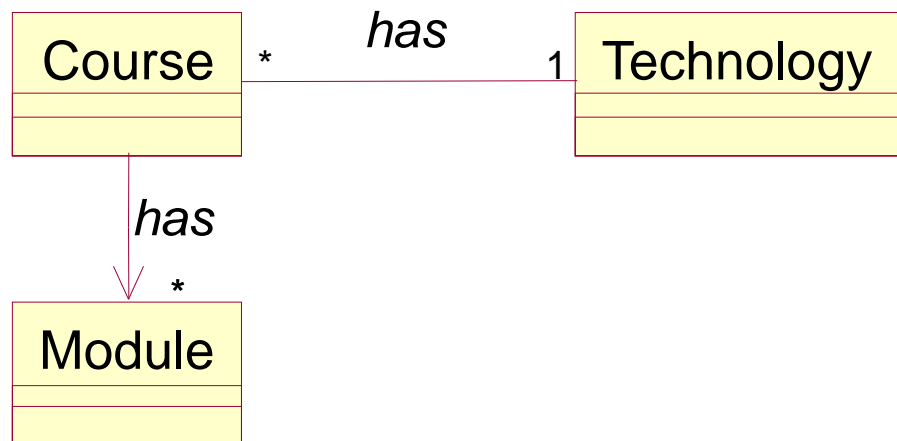
Module		

Identifying Relationships

- Module – Unit
  - Module 'HAS' many Units
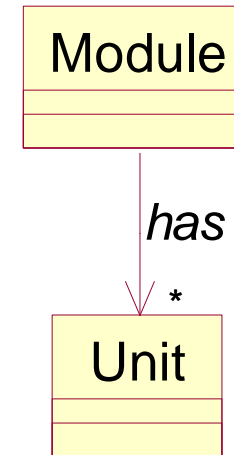
- Unit – Topic
  - Unit 'HAS' many Topics

The OO Model

Trainer			has			Trainee	

Trainer (1) has Trainee (*)

Course (*) has (1) Technology

Course has (*) Module

Module has (*) Unit

Unit has Topic (*)
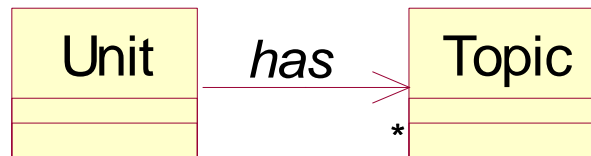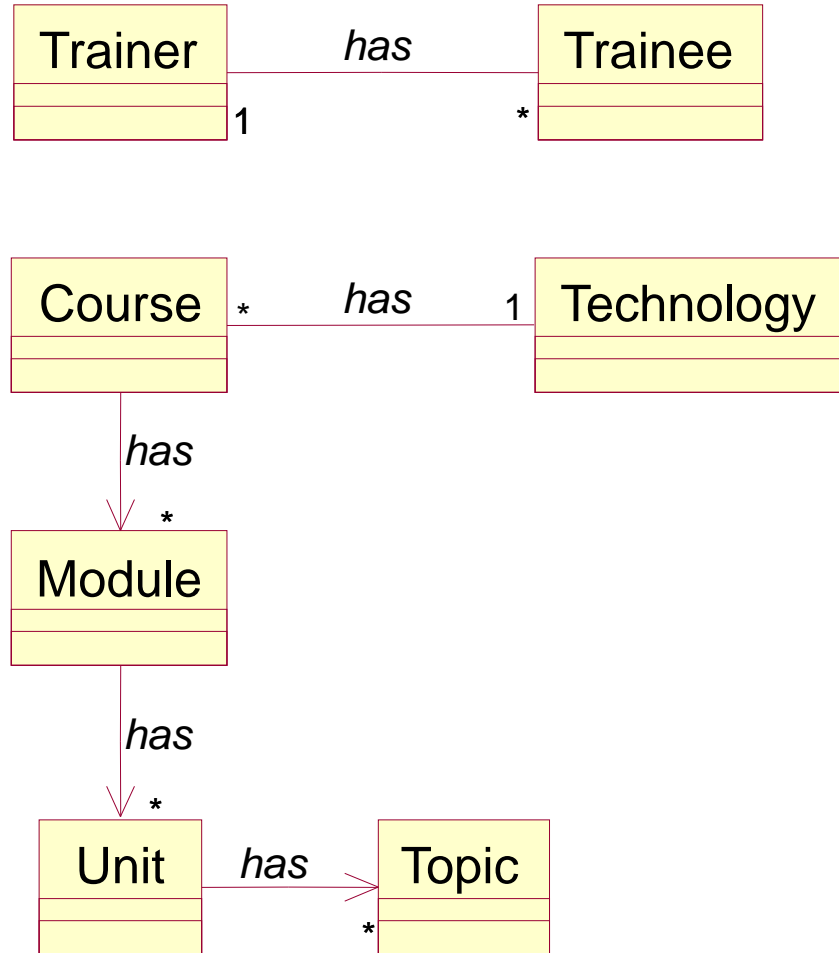
How do you relate the Trainer & Trainee to the Course?

Conceptual Entity

- Trainer – Training
  - A Trainer (HAS) conducts many Trainings
  - A Training HAS a Trainer

Trainer

1

has

Training

*
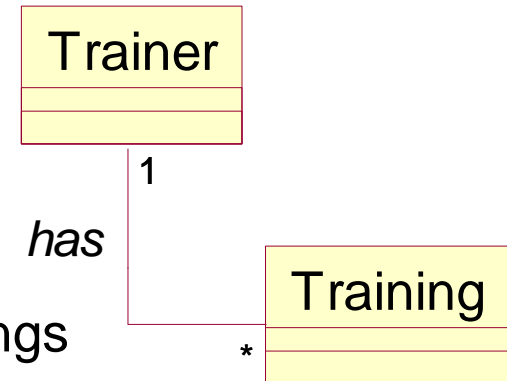
- Trainee – Training
  - A Trainee (HAS) attends many Trainings
  - A Training HAS a many Trainees

Trainee

*

has

Training

*
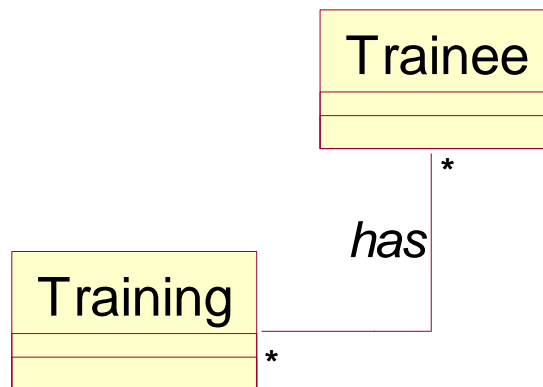
Conceptual Entity

- Training - Course
    - The Training (HAS) an association with a Course (conducted for a Course)
    - A Course HAS many Trainings

```
┌─────────────────┐
│    Training     │
├─────────────────┤
│                 │
├─────────────────┤
│                 │
└─────────────────┘
         *
        has
         1
┌─────────────────┐
│    Course       │
├─────────────────┤
│                 │
├─────────────────┤
│                 │
└─────────────────┘
```
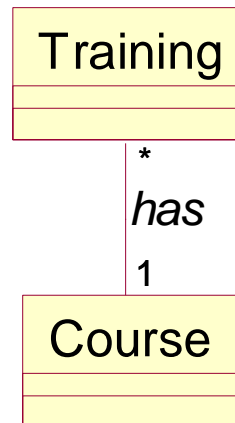
Solution
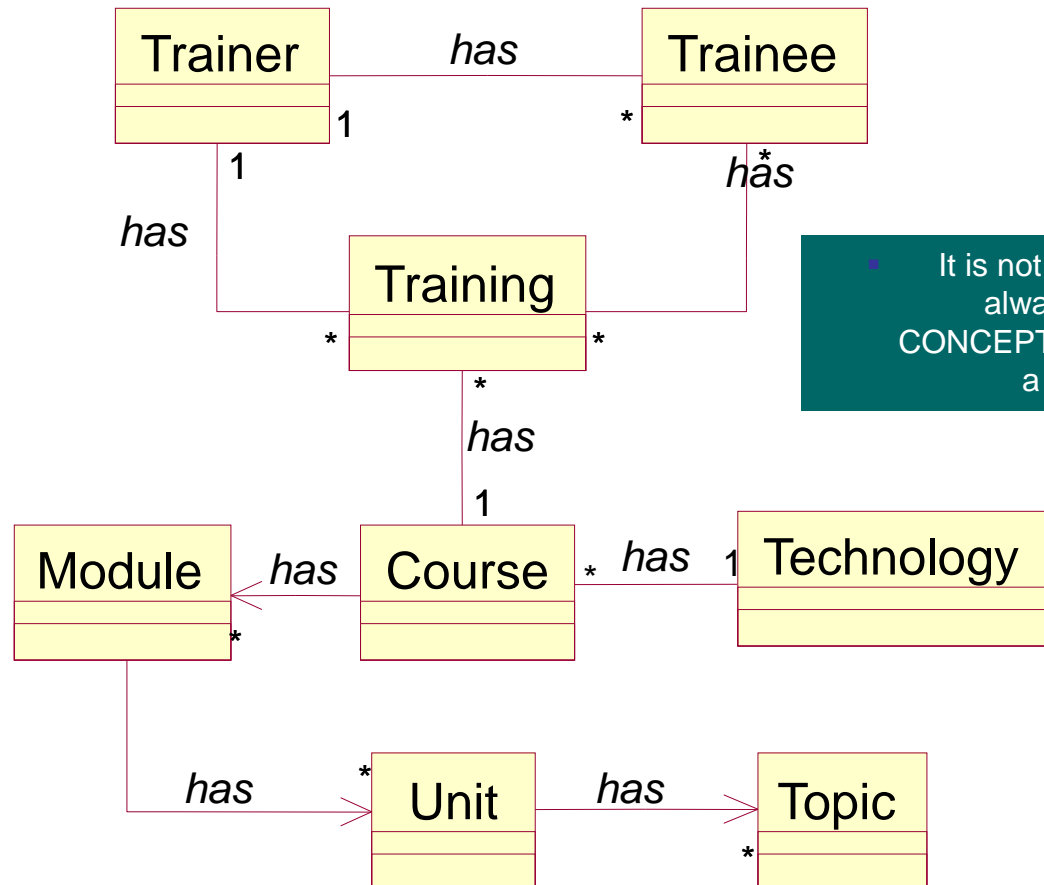
It is not necessary to always have a CONCEPTUAL ENTITY in a Design

Easier to model real-world problems through the OO approach than through the procedural approach

Exercise

A company sells different items to customers who have placed orders. An order can be placed for several items. However, a company gives special discounts to its registered customers.

- Identify the different classes from the above problem statement
- Identify the different connections (relationships) between the above classes

Identifying Classes

- Company
- Item
- Order
- Customer
- RegCustomer

Identifying Relationships

- Company - Customer
    - Company 'HAS' many Customers
    - Customer 'HAS' many Companies
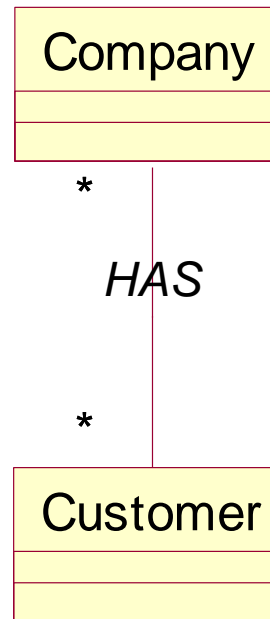
Identifying Relationships

- Company - Item
  - Company HAS many Items

Company			HAS →	Item		
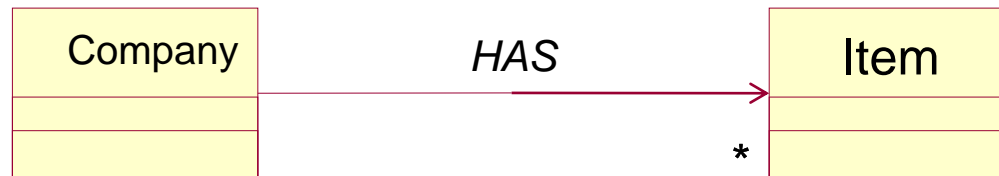
Identifying Relationships

- ## Customer - RegCustomer
  - ### RegCustomer 'IS' a Customer

```
┌─────────────────┐
│    Customer     │
├─────────────────┤
│                 │
├─────────────────┤
│                 │
└─────────────────┘
         △
         │
       IS a
         │
┌─────────────────┐
│   RegCustomer   │
├─────────────────┤
│ discount        │
├─────────────────┤
│                 │
└─────────────────┘
```
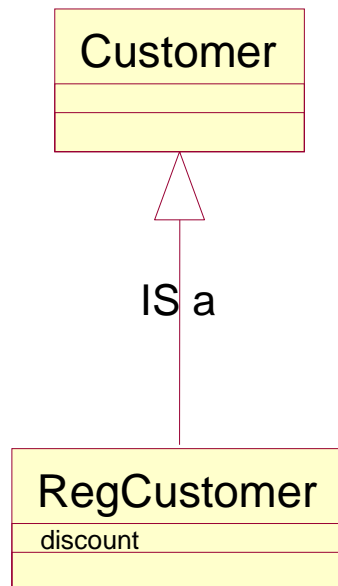
Identifying Relationships

- ## Order - Item
    - ### Order HAS many Items

Order		HAS	Item	
		*		
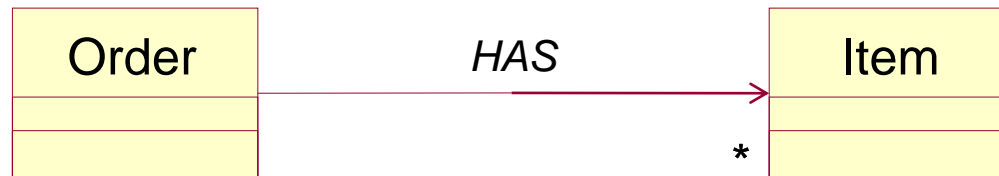
Identifying Relationships

- Customer - Order
  - Customer HAS many Orders
  - Order HAS one Customer

```
┌─────────────────┐
│    Customer     │
├─────────────────┤
│                 │
├─────────────────┤
│                 │
└─────────────────┘
         │ 1
        HAS


         │ *
┌─────────────────┐
│     Order       │
├─────────────────┤
│                 │
├─────────────────┤
│                 │
└─────────────────┘
```

The OO Model

```
         Company
         ┌──────────┐
         │          │
         └──────────┘
             *
             │
           HAS
             │
             *
         Customer ◁──────── IS a ──────── RegCustomer
         ┌──────────┐                     ┌──────────┐
         │          │                     │ discount │
         └──────────┘                     └──────────┘
             1
           HAS
             │
             *
          Order ──── HAS ────▶  Item
         ┌──────┐              ┌──────┐
         │      │ *            │      │ *      *
         └──────┘              └──────┘
```
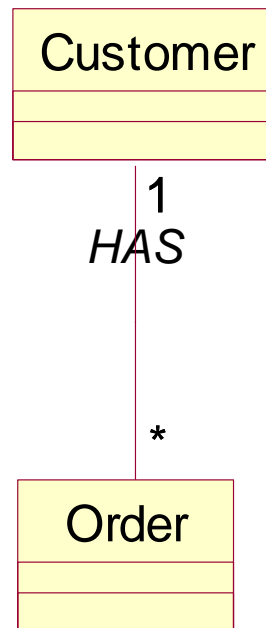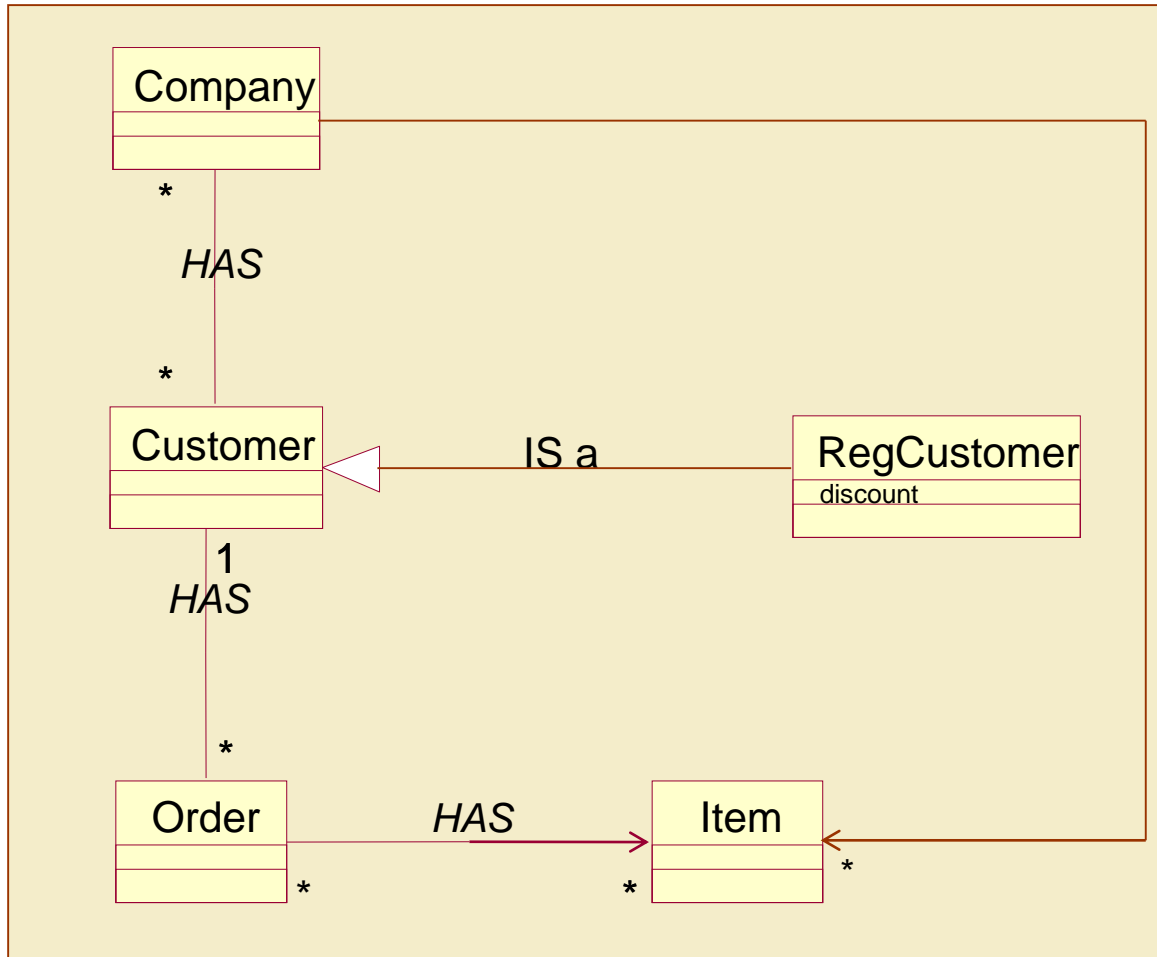
A Customer can place many orders implies that RegCustomer can also place many Orders.

A Company has many Customers implies that a Company also has many RegCustomers

Basic Java

Class in Java

- A *class* is a software construct that defines the instance variables and methods of an object.

- A *class* is a template that defines how an object will look and behave when the object is created or instantiated from the specification declared by the class.

- A *class* can be viewed as a user defined data type.

```java
class Point
{
    double x;
    double y;

    double getX()
    {
        return x;
    }
}
```

Structure of a class

```
class Employee
{
        String employeeId;
        String employeeName;                                    Instance Variables

        Employee()
        {
                System.out.println("Constructor called");       Constructor
        }

        void setEmployeeId(String employeeId)
        {
                this.employeeId = employeeId;
        }
        String getEmployeeId()
        {                                                        Methods
                return employeeId;
        }
}
```

What is an Object?

- An object is an entity with a well-defined *State* and *Behavior*

- An *object* is created from the class definition using the *new* operator.

- The state of an object is referred to as the values held inside the instance *variables* of the class.

- The behavior of the class is referred to as the *methods* of the class.

- To create an object of the class Point, say,
  Point p = new Point();

- When an object of the class is created, memory is allocated for all the instance variables, here p is not an object but a *reference* or *handle* to an object being created.

```java
class Point
{
        double x;
        double y;

        double getX()
        {
                return x;
        }
}
```

Instantiating Classes

```
public class Shop

                      void main(Stri...
{
          Product p1=new Product();
          p1.id=1;
          p1.name="Steam Iron";

          Product p2=new Product();
          p2.id=2;
          p2.name="Microwave"

          p1.makePurchase();
          ................
     }
}
```

P1 is a reference

The RHS creates an instance

Exercise

- Create a class Employee with the following data members and methods
  - Data
    - empID : string
    - empName : string
  - Methods
    - Set methods and get methods for the data members

- Write a class EmployeeData with two methods
  - storeData() which takes the Employee object as an argument and returns the same
    - Accepts user input for employee data and sets the data on the object
  - showData() which takes an Employee object as argument and displays the data from the object

- Write a class EmployeeDemo with the main method
  - Create an instance of the Employee object and pass the same to the storeData() and showData()

Modeling the 'has-a' relationship
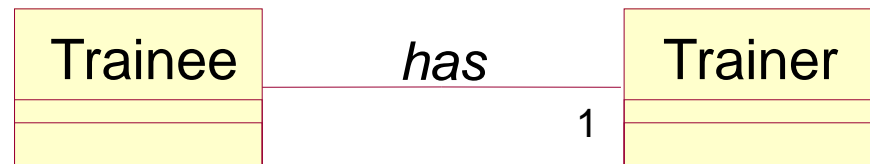
Trainee   *has*   Trainer
                        1

```java
public class Trainee {
    String department;
    String yrs_of_experience;

    Trainer t;
}
```

```java
class Trainer {
    String vendorName;
    String[] techStack;
    String[] monthly_schedule;
}
```

Basic Java

Exercise

- Write a class Address with the following data members and methods
    - Data
        - addr1 : string
        - addr2 : string
        - city : string
        - pin : int
    - Methods
        - set and get methods

- The relationship between Employee class and Address class is EMPLOYEE has ADDRESS
    - Model the relationship

- Modify the storeData methods of EmployeeDate class to also accept and include Address details

Modeling 'has-a' with multiplicity 'n'

Trainer	*has*	Trainee

1   *

```
class Trainer {
    String vendorName;
    String[] techStack;
    String[] monthly_schedule;

    int num_of_trainees;
    Trainee[] trainees = new Trainee[10];
}
```

Trainer.java   Trainee.java

```
public class Trainee {
    String department;
    String yrs_of_experience;

    Trainer t;
}
```

Trainer.java   Trainee.java

Basic Java
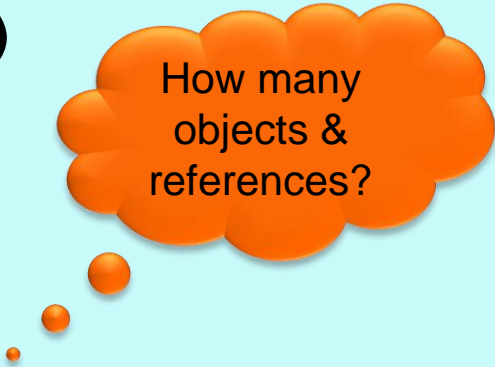
References v/s Objects

```java
public class Shop
{
    public static void main(String[] args)
    {
        Product p1=new Product();
        p1.id=1;
        p1.name="Steam Iron";

        Product p2=p1;
        p2.id=2;
        p2.name="Microwave";
    }
}
```
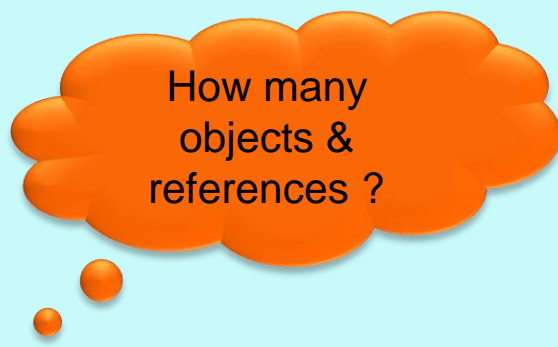
How many objects & references?

References v/s Objects

```java
class Person {
    private String name;
    private int age;
    Person(String n, int a) {
        name = n;    age = a;
    }
    public void printPerson() {
        System.out.println("Hi, my name is " + name);
        System.out.println(". I am " + age + " years old.");
    }
    public static void main (String args[]) {
        Person p1;
        p1 = new Person("Luke", 50);
        p1.printPerson();
        p1 = new Person("Laura", 35);
        p1.printPerson();
    }
}
```
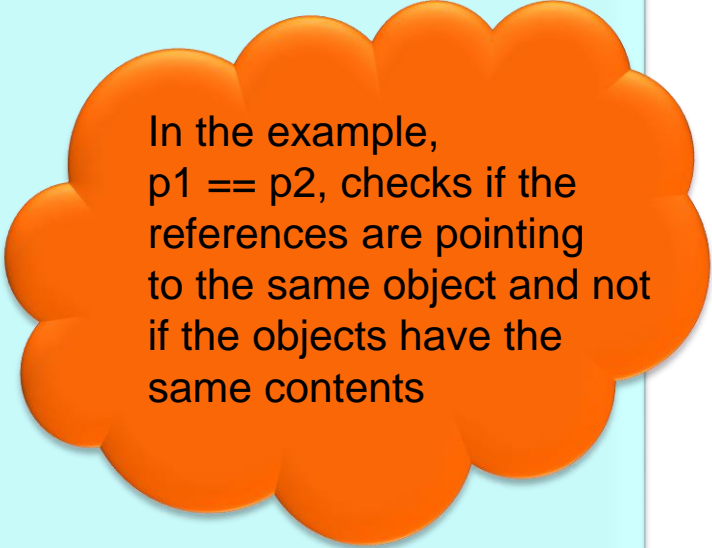
How many objects & references ?

Comparing Objects

```java
class Point
{
        private double x;
        private double y;
        public Point(int _x , int _y){
                x = _x;
                y = -y;
        }
        public double getX()
        {
                return x;
        }
        public static void main(String[] args)
        {
                Point p1 = new Point(10, 20);
                Point p2 = new Point(10, 20);
                if(p1 == p2)
                        System.out.println("The two objects have the same content");
                else
                        System.out.println("The two objects have different content");
        }
}
```

In the example, p1 == p2, checks if the references are pointing to the same object and not if the objects have the same contents

Comparing Objects

- The '==' operator when used with objects, does not compare the states of the objects.

- Instead it compares whether the two references point to same object in memory or not.

- It is simply because the compiler does not know how to compare user defined types, Eg., how can the compiler know how to compare 2 customers (i.e., objects of Customer class)

- To do more meaningful comparison, the equals method is used.

  - The programmer is responsible for providing this method for his classes

Initialising Objects

- All data members in a class can be initialised at the point of declaration in a class.

- If the primitive types are not initialised they are default set to 0 for numeric, set to ' '(whitespace) for char and set to false for boolean data type.

- Similarly a reference type can be initialised at the point of declaration in a class, if not they are set to null.

- Data members are initialised before any method or constructor is called.

- **What is a constructor?**

See listing : **InitialisationDemo.java**

Constructor

- Constructor is a special method with the same name as that of the class.

- Constructors are called implicitly at the time of object creation.

- Constructors do not have a return type.

- Every class has at least one constructor.
  - Either defined by programmer or the compiler provides a default constructor.

```
class Product
{
        Product()
        {
                //instantiate the
                variables
                ……………..
                ………………
        }
}
```

Why Constructors?

```java
public class Shop
{
    public static void main(String[] args)
    {
        Product p1 = new Prod
        p1.id = 1;
        p1.name = "Steam Iron",
        p1.price = 100;
        p1.category = "luxury";

        if (p1.price > 99  &&  p1.category.equals("luxury"))
            p1.tax=p1.price*0.20;
        else
            p1.tax=p1.price*0.10;
    }
}
```

Here, the tax is calculated based on the price and category. What if the programmer forgets to calculate tax?

Why Constructors?

```java
class Product
{

    int productId;
    String productName,category;
    float price,tax;
    Product(int id , String name ,
                    String cat , float p)

    {

        productId = id;
        productName = name;
        category = cat;
        price = p;
        if (price>99 && category.equals("luxury"))
            tax = price*0.20;
        else
            tax = price*0.10;
    }
```

```java
public static void main(String[] args){
Product p1 = new Product(1,"Steam Iron","luxury",100);
p1.makePurchase();
}
```

Compiler and Constructors

```
class Product
{
    int id;
    String name;

    public void changePrice()
    {
        …………..
    }
}
```

```
public class Product
{
    int id;
    String name;
    public Product(int id){
    ………..
    }
    public void changePrice()
    {
        …………..
    }
}
```

No constructor defined by the programmer. The compiler adds the public no-args constructor

A constructor defined by programmer. Compiler does not add anything

Basic Java

Exercise

- Write a class Rectangle with data members, double length and double breadth. Write a parameterized constructor which takes length and breadth as parameters and assigns them to the instance variables. Write a method area() which calculates the area and returns the same.

- Write a class InterestCalculator with data members, double principle , int time , double intRate. Write a parameterized constructor which takes principle and time. The intRate has to be calculated in the constructor based on the time, if time is >= 5 the rate is 10% else 12%. Write a method getInterest() which calculates the simple interest and returns the same. Test the code.

Class Creator Vs. Class User

- To truly appreciate many concepts of OOP, it is important to distinguish between creators of the class and users of the class.

- Often, the creators of the class are not the users of the class.
  - Eg, there are tons of classes in Java's built-in libraries authored by creators of the language but are used by us.

- Do not get confused between class user and end users.

- Class users are also programmers who are creating their program using the classes provided by the class creators
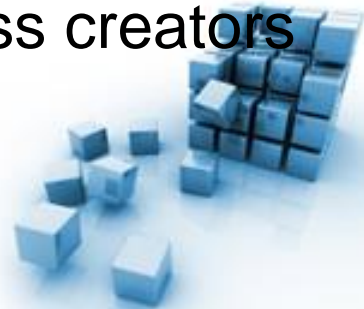
Abstraction

Public View of an Object
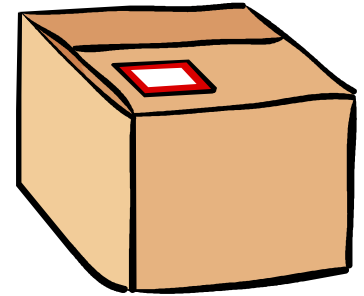
- *Abstraction* is used to manage complexity
  - Focus on the essential characteristics
  - Eliminate the details
  - Find commonalities among objects

- Defines the public contract
  - Public definition for users of the object
  - The "Outside view"
  - Independent of implementation

Abstraction

Public View of an Object
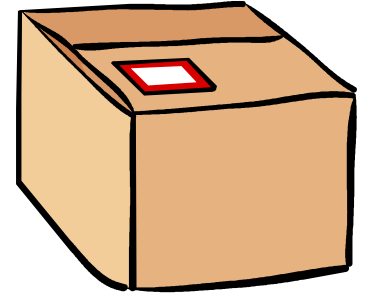
- *Abstraction* is used to minimize complexity for the class user
  - By allowing him focus on the essential characteristics
  - By hiding the details of implementation

- Simply put, abstraction is nothing but a process of ensuring that class users are not exposed to details which they do not need (or use).

Abstraction

ABSTRACTION:

- What should the Cell Phone do?
- What are the responsibilities of the Cell Phone?

Basic Java

Abstraction - Example

Order

CreateOrder
UpdateOrder
GetOrderTotal
Next

"What should an Order object do?"

Two types of Orders

InternalOrder

ExternalOrder
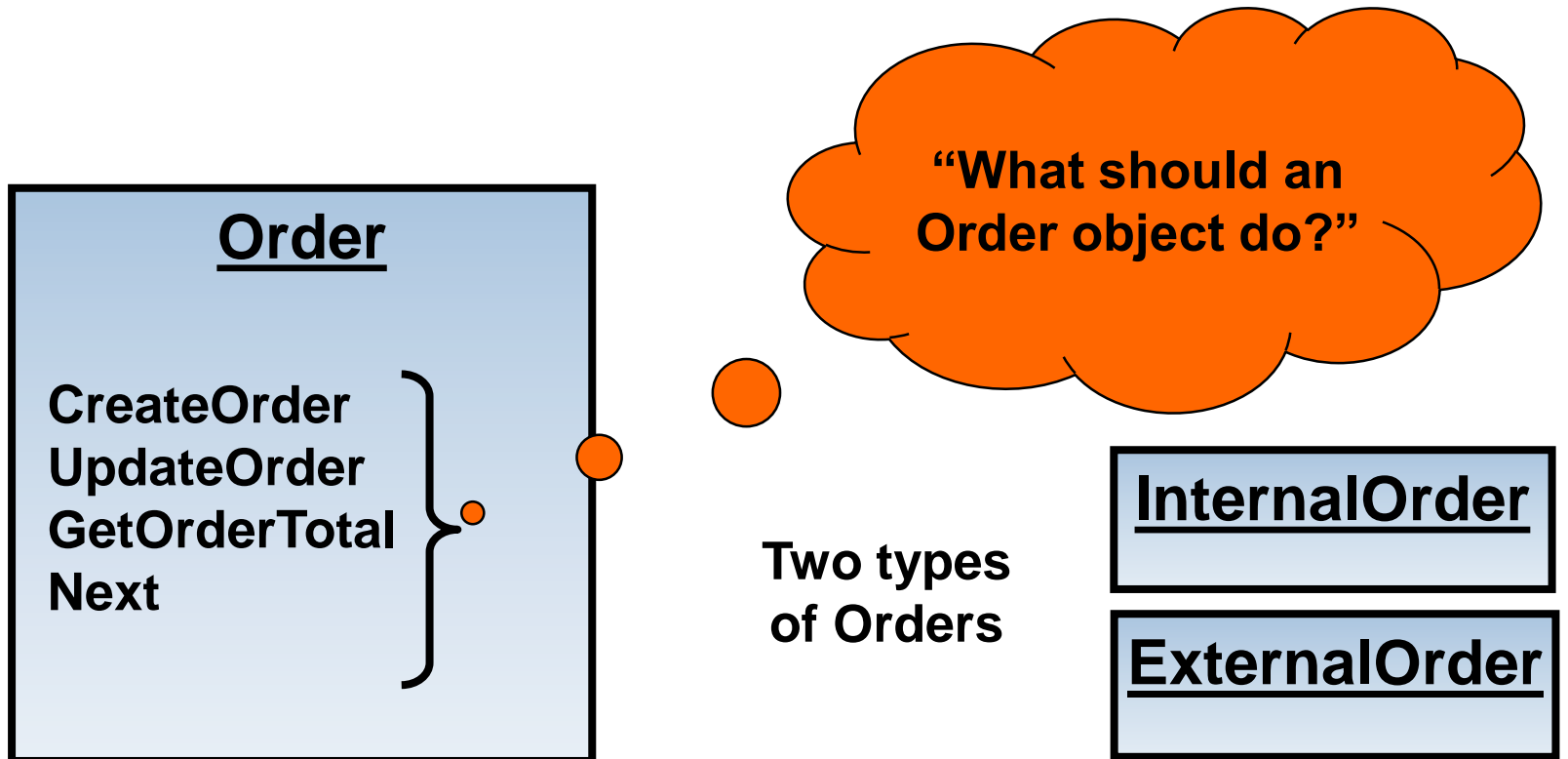
Encapsulation

Hide Implementation Details

Employee
empId : String
name : String
address : Address
getEmpID() : String
setEmpId(empId : String)
getName() : String
setName(name : String)
getAddress() : Address
setAddress(address : Address)

- Encapsulation is
  - The grouping of related ideas into a single unit, which can thereafter be referred to by a single name.
  - The process of compartmentalizing 'the elements of an abstraction' that constitute its **structure** and **behavior**.

- *Encapsulation* hides implementation
  - Promotes modular software design – data and methods together
  - Data access always done through methods
  - Often called "information hiding"

- Provides two kinds of protection:
  - State cannot be changed directly from outside
  - Implementation can change without affecting users of the object
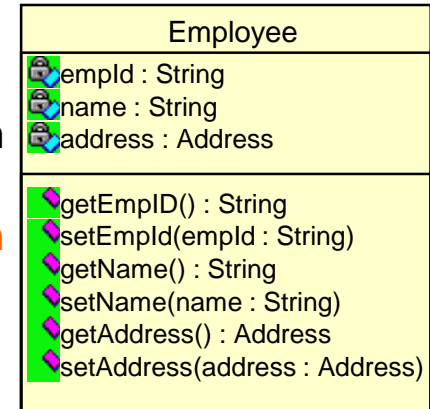
Encapsulation

ENCAPSULATION:

- What all should the Cell Phone contain (encapsulate) to meet its responsibilities?
- What are all needed to provide an implementation for the ABSTRACTIONS?

Encapsulation - Example

Implementation

Outside View

Order

orderNum AS INT
custNum   AS INT
CalculatePrice( )

PUBLIC:
CreateOrder( )
UpdateOrder( )
GetOrderTotal( )
Next( )

**Public methods
of Order**

CreateOrder
UpdateOrder
GetOrderTotal
Next

Encapsulation - Example continued

Order

orderNum AS INT
custNum   AS INT
CalculatePrice( )

PUBLIC:
CreateOrder( )
UpdateOrder( )
GetOrderTotal( )
Next( )

Hmm...
I'd like to change
CalculatePrice to
CalculateTotalPrice

GetOrderTotal calls
CalculatePrice( )

Encapsulation - Example continued

Order

orderNum AS INT
custNum   AS INT
CalculateTotalPrice( )

PUBLIC:
CreateOrder( )
UpdateOrder( )
GetOrderTotal( )
Next( )
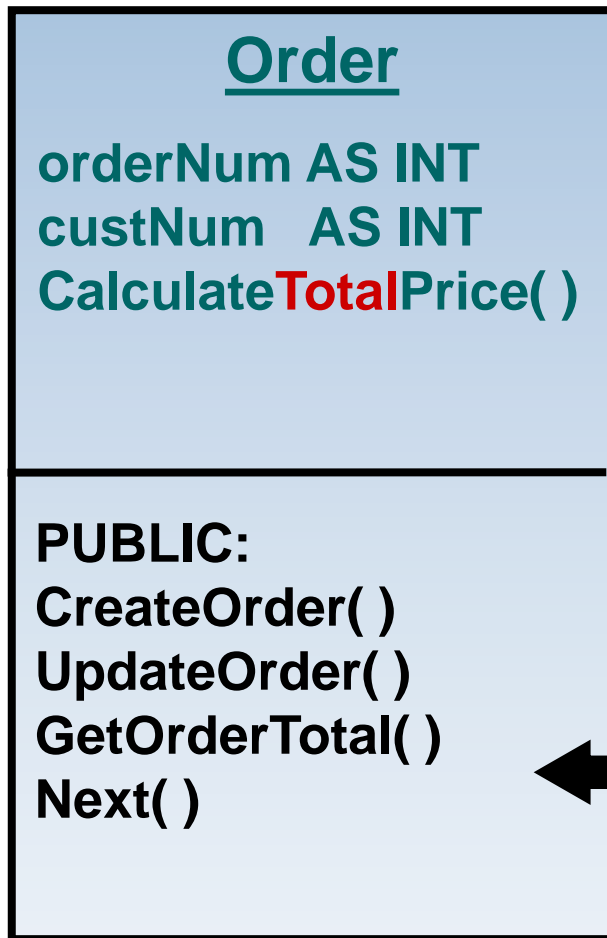
This change was easy because users of the object will not be affected.

GetOrderTotal now calls CalculateTotalPrice( )

Implementing Abstraction and Encapsulation

class Order

{

int orderNum;

int custNum;

double calculateTotalPrice(){..........}

Invisible

void createOrder() {........... }   **Visible**

void updateOrder() {...............}

double getOrderTotal() { ………. }

Order next() { ………… }

}

What should be exposed to the class user?

Differentiating Abstraction & Encapsulation

ABSTRACTION:

- What should the Cell Phone do?
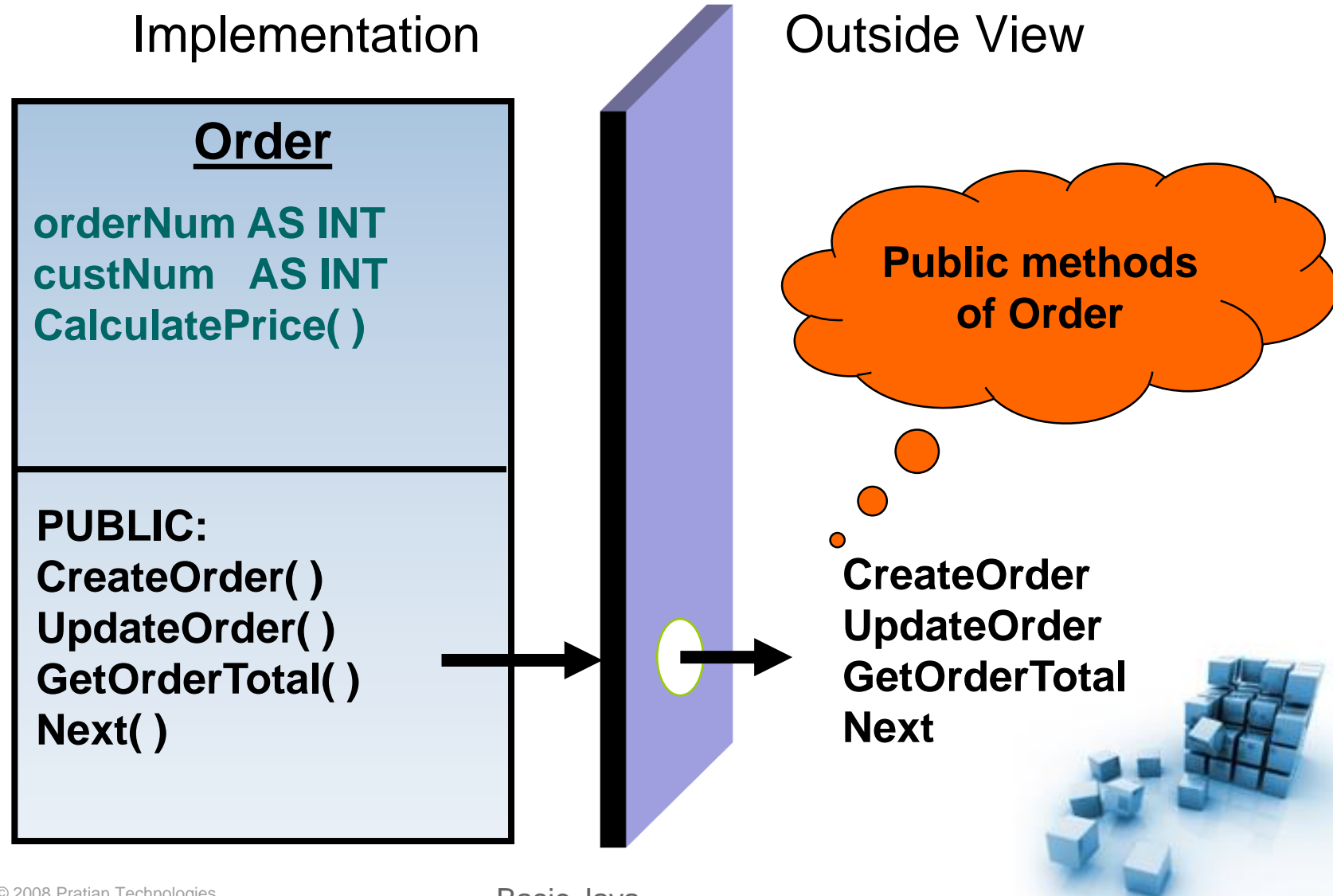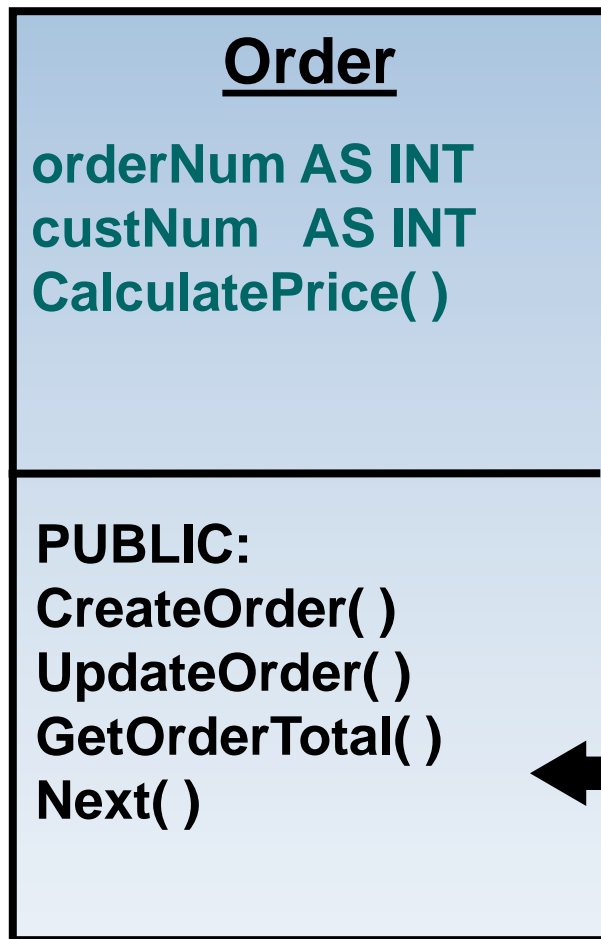- What are the responsibilities of the Cell Phone?

ENCAPSULATION:

- What all should the Cell Phone contain (encapsulate) to meet its responsibilities?
- What are all needed to provide an implementation for the ABSTRACTIONS?

ic Java

Access Specifiers

- Access modifiers are those which control access to methods and variables. *public , private, protected and default*

- public
  - Any class member declared as public is visible (or accessible) to the whole world (meaning any class)

- private
  - Any class member declared as private is visible (or accessible) only inside the same class

- protected *(More on this later)*
  - Any class member declared as protected is visible (or accessible) to all classes in the same package as well as to sub classes (regardless of the package)

- default *(More on this later)*
  - Any class member declared without any of the above is visible (or accessible) to all classes in the same package only.

Specifying Access

```java
class Order
{
        private int orderNum;
        private int custNum;
        private double calculateTotalPrice(){..........}


        public void createOrder() {  ...........  }

        public void updateOrder()   {...............}

        public double getOrderTotal() { ……….. }

        public Order next() { ………… }
}
```

Exercise

Let us say we have to design a class Car to represent the automobile in our domain.
Assume some requirements and realize the abstractions and encapsulations of the car.
Apply access specifiers to provide the right abstraction to the users of the class

Implementing Abstraction & Encapsulation

```java
public class Car
{
    private Engine e;
    private FuelTank tank;
    private void pullFuelFromTank(){..........}
    private void regulateEngineTemperature(){.........}


    public void start()
    {

            ............
    }
    public void stop()
    {

            ...............
    }
}
```

If you are wondering…

- If you are wondering what's the difference between Abstraction and encapsulation

- Encapsulation deals with '**what goes into**' a class

- Abstraction deals with **'what is made visible'** to the class user.

Static Members

- Methods and variables of a class can be marked as **_Static_**

- Static members are not tied to any instance of the class, rather they are termed as the class members.

- The static members of the class can be accessed directly without creating an instance of the class.

- Static methods cannot access non-static members.

- But non-static members can access static members.

```java
public class Product
{

    int id;
    String name;
    static int count;
    public Product(int id ,
                    String name)
    {
        this.id = id;
        this.name = name;
        ++count;
    }
    public static int getCount(){
        return count;
    }
}
```

See listing : **StaticMembersDemo.java**

Static Methods - Example

```java
class Factorial {
    public static void main(String[] args) {
        int input = Integer.parseInt(args[0]);
        double result = calculateFactorial(input);
        System.out.println("Factorial of  "+input+
                        "is: "+(int)result);
    }

    public static double calculateFactorial(int x) {
        if (x < 0)
                return 0;
        double fact = 1;
        while (x > 1) {
                fact = fact * x;
                x = x - 1;
        }
    return fact;
    }
}
```

Static Initialization

- Static data members can be initialized inside a static construction clause(*static block*) and it happens only once when the class is loaded in memory.

```java
public class BillingSys
{
    Product[ ] productList;
    static int taxRate;
    static
    {
        taxRate = open a configuration file and
                    read from the file
    }
    public static int getTaxRate(){
        return taxRate;
    }
}
```

Inheritance

- An object oriented system organizes classes into a subclass-super class hierarchy
    - Inheritance encourages 'code reuse'

- **Each subclass reuses the implementations in the base class**
    - **Can add new responsibilities**

Generalization

Specialization

Mary

Employee

Manager   Salesperson
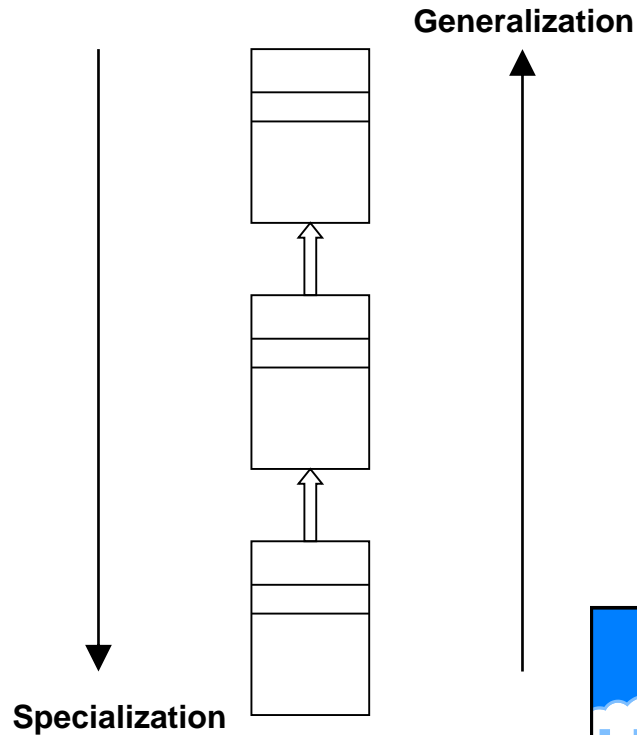
David
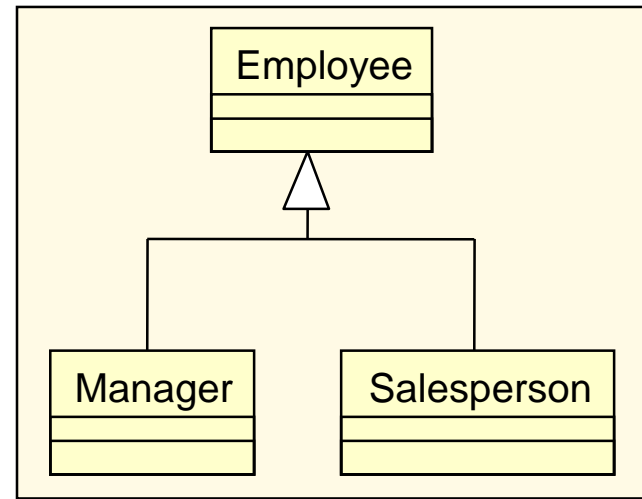
Joe

Why Inheritance?

```java
public class Car
{
    Engine e;
    FuelTank tank;
    void pullFuelFromTank()
    void regulateEngTemp()
    void start()
    void stop()

    switchOnAC()
}
```

Redundant!!

```java
public class Truck
{
    Engine e;
    FuelTank tank;
    void pullFuelFromTank()
    void regulateEngTemp()
    void start()
    void stop()

    loadGoods()
    unloadGoods()
}
```

Inheritance for Reuse

```java
public class Vehicle
{

    Engine e;
    FuelTank tank;
    void pullFuelFromTank()
    void regulateEngineTemperature()
    void start()
    void stop()

}
```

```java
public class Car extends Vehicle
{
    switchOnAC()
}
```

```java
public class Truck  extends Vehicle
{
    loadGoods()
    unloadGoods()
}
```

Inheritance and Access Specification
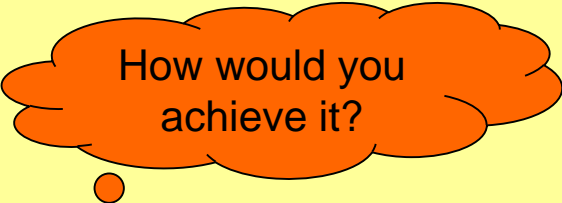
public class Vehicle

Should not be visible to class users but should be visible to subclasses

{

Engine e;

FuelTank tank;

void pullFuelFromTank(){..........}

void regulateEngineTemperature(){.........}

How would you achieve it?

Should be visible to everyone

void start(){...........}

void stop(){...............}

}

Inheritance and Access Specification

```java
public class Vehicle
{
    protected Engine e;

    protected FuelTank tank;

    protected void pullFuelFromTank()

    protected void regulateEngineTemp()


    public void start()

    public void stop()

}
```

Should not be visible to class users but should be visible to subclasses

Should be visible to everyone

Object Class

- Class Object is the root of the class hierarchy in Java.

- All objects either directly or indirectly inherit from this class.

- Some important methods of the class

  - protected Object clone()

  - public boolean equals(Object obj)

  - protected void finalize()

  - public String toString()

  - public void notify()

  - public void wait()

Constructor Chaining

- Every constructor method calls its base class constructor directly using *super()* or indirectly using *this()*.

- If the first statement of a constructor does not explicitly call *this()* or *super()*, the compiler adds the call to the default super constructor

  - If super class does not have a default constructor, compiler will throw an error

- So, always , whenever any object is created, the very first constructor to fully get executed is of java.lang.Object

  See listing : **ConstructorChaining.java**

Exercise

- Write a class Customer with the following data members,

  String : custId

  String : name

  Address : address

  Write a parameterized constructor which takes custId , name and address as input and assigns the same to the instance variables and write getter methods for all the instance variables.

- Write a class Address with the following data members,

  String : addr1

  String : addr2

  String : city

  int : pin

  Write the respective setter and getter methods.

- Write a class RegCustomer which extends from the class Customer and has the instance variable

  double : fees.

  Write a appropriate parameterized constructor and call the base class constructor.

- In the main create an instance of the class RegCustomer by passing the values and display the same.

Basic Java

Type Casting of Primitives

- A primitive of one data type can be cast to other type in Java.
  - Casting is possible if the the two types are compatible.
    - All numeric types are compatible with each other.
    - Integers are compatible with characters.
    - Boolean is not compatible with any of the data type.
  - Casting is implicit if destination type is larger than source type.
    - Eg : int to double;

      int to long, short to int.
  - Casting needs to be explicit if the destination type is smaller than source type.This may lead to loss of data.
    - Eg : double to int;

      long to int;

See listing : **PrimitiveTypeCast.java**

Type Casting of Objects

- Objects can be typecast only if they are related by inheritance not otherwise.

- A derived class object can be automatically typecast to a base class reference.

  - eg

    Employee emp = new Manager();//implicit

    Manager mgr = new Manager();

    mgr = (Manager) emp; // works //explicit

    emp = mgr; // works

Comparing Objects

- The '==' operator when used with objects, does not compare the states of the objects.

- This logic of comparing two objects of a class is best known to the author of the class

- To do more meaningful comparison, the equals method is used.
    - The programmer is responsible for providing this method for his classes

Comparing Objects

```java
class Point {
    private double x , y;
    public Point(int _x , int _y){
        x = _x;
        y = -y;
    }
    public boolean equals(Object o) {
        Point p = (Point) o;
        if(p.x == x && p.y == y)
            return true;
        else
            return false;
    }
    public static void main(String[] args){
        Point p1 = new Point(10,2
        Point p2 = new Point(10,20);
        if(p1.equals(p2))
            System.out.println("Objects are duplicates, with the same
                            contents");
        else
            System.out.println("Objects have different contents");
    }
}
```

p1.equals(p2) checks if the two objects are same by checking the contents .
The equals() method is overridden with the comparison logic

See listing : **Point.java**

Polymorphism

One function, many implementations

All employees do some work

Mary

Employee

◆work()

David

Manager | Salesperson
◆work() | ◆work()
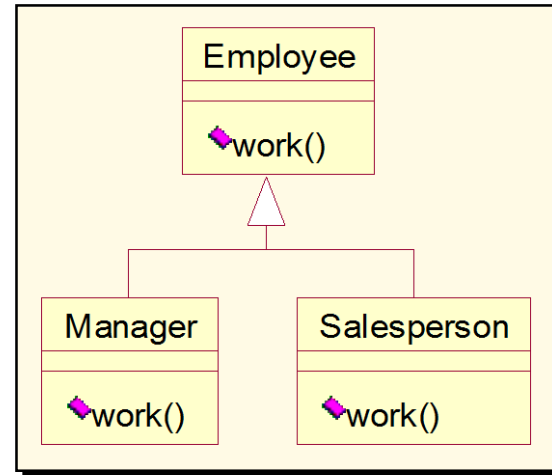
Joe

David does a manager's work

Joe does a salesperson's work

- Early binding
  - Function Call mapped at compilation
  - Function Overloading

- Late binding
  - Function Call mapped at run-time
  - Function Overriding

▪Runtime Polymorphism (late binding) has three requirements:

- ▪ Hierarchy with **overridden** method in derived class

- ▪ Base class reference used to call method

- ▪ Derived class assigned to base class reference

Basic Java

Polymorphism

- Two types of polymorphism

  - Static polymorphism

    - Method overloading

    - Input data (parameter) determines the type of method to be called

  - Dynamic polymorphism

    - Method Overriding

    - Type of object pointed by an interface or a super class variable determines the specific action

Method Overloading

- Overloading is achieved by having multiple methods with the same name but with different parameters.

- Multiple methods with the same name differ based on the parameters.

- It is easier for the class users to remember fewer number of method names

- Methods cannot be overloaded based on the return type.

```java
public class Addition
{

    public int add(double x ,
                    double y)
    {
        return x + y;
    }
    public int add(int x , int y)
    {
        return x + y;
    }
}
```

See listing : **OverLoadingDemo.java**

Overloaded Constructor

```java
class Employee {
    String empId,empName;
    double salary;
    //overloaded constructor
    Employee() {}
    Employee(String id,String name,double sal)
    {
        empId = id; empName = name; salary = sal;
    }
    public static void main(String ars[])
    {
        Employee e1 = new Employee();
        Employee e2= new Employee ("951002","Sam",23480);
    }
}
```

Method Overriding

Vehicle
start()
stop()

Bike
start()
stop()

Car
start()
stop()

Bus
start()
stop()
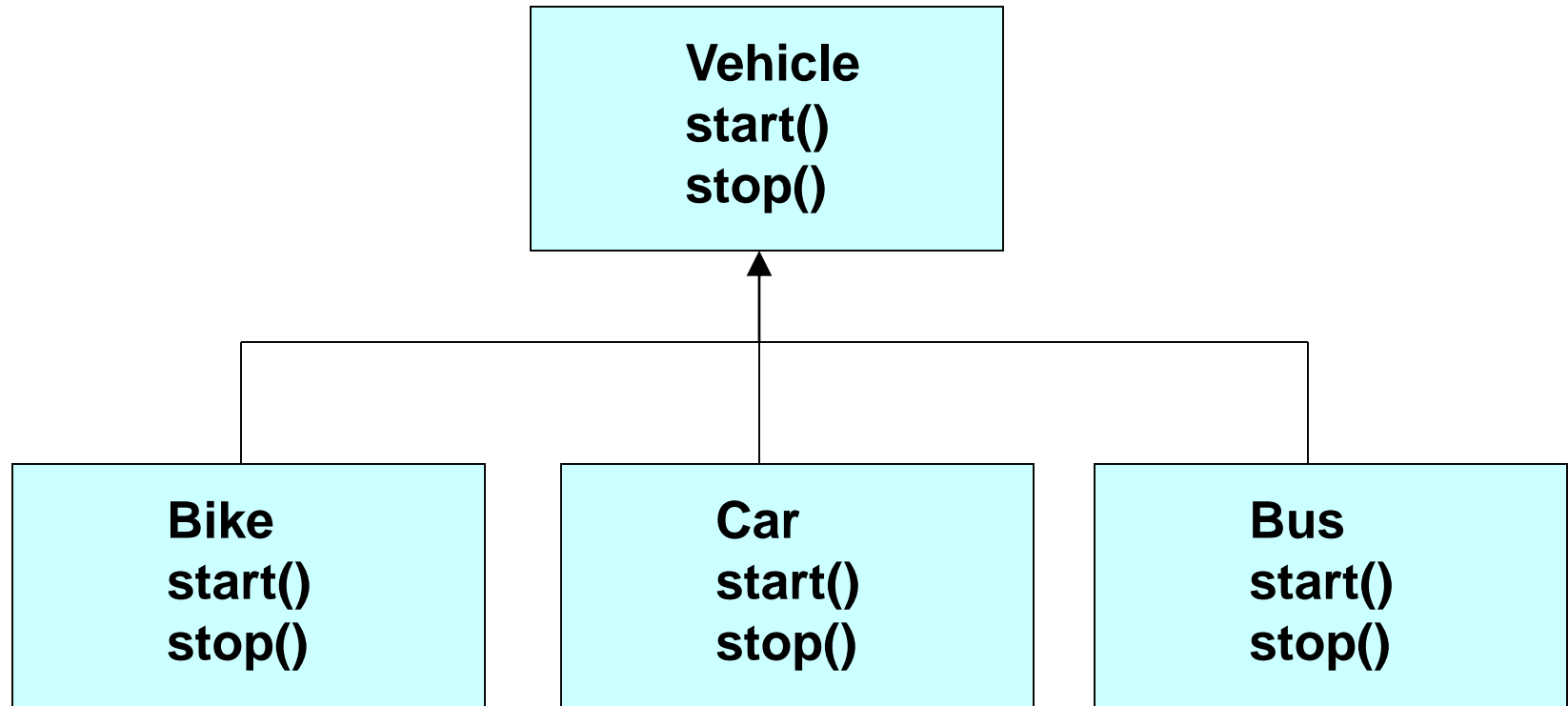
See listing : **OverridingDemo1.java**

Method Overriding

- Redefine the method in the subclasses with the same signature as a method in the superclass

- Used when the behaviour of the child class is different from that of the base class

- Method in sub class overrides the method in the superclass

- Methods cannot be overridden to be more private, only to be more public

See listing : **OverridingDemo2.java**

Overloading v/s Overriding

Methods in same class	Methods in superclass and subclass
Method Signature is different	Method Signature has to be same
The Parameters decides which method to call	The Object decides whether to call parent or child class method
Constructors can be Overloaded	Constructors cannot be Overridden

Exercise

- Write a class Addition, which has a method add() overloaded to add two Strings, ints, double and test the code.

- Write a class Person with data members

  String name

  int age

  char sex

  and write appropriate setter and getter methods.

  Write the equals() method to check for equality

  Create two instances of the Person class and pass

  the data. Check if the two person instances are

  same by using the equals() method and display the

  same.

Final Modifier

- The *final* modifier is used with variables, methods and classes to indicate they cannot be changed.

- The value of a data member marked as final cannot be changed after initialization

- Methods marked as final cannot be overridden in its child class.

- Classes marked as final cannot be sub classed.

Inner Classes

- Java allows a class to be defined within another class, such a class is called a *inner class*.

- Inner classes have access to variables and methods of the enclosing class.

- As a member of outerclass, a inner class can be declared with public, private, protected static or default access specifiers/modifiers.

```
public class OuterClass
{

       ................
       ................

       class InnerClass
       {

              ...........
              ...........
       }

}
```

Why use Inner Classes?

- *Logical grouping of classes* – If a class is useful to only one other class, then it is logical to embed it in that class and keep the two together. Nesting such *helper classes* makes the package more streamlined.

- *Increased encapsulation* – Consider two classes A and B, here B a helper class needs access to private members of class A. Here B can be nested inside class A, thereby access the private members of A and B is hidden from outside world.

- *More readable, maintainable code* – Nesting small classes within top-level classes places the code closer to where its used, thereby its better readable and maintainable.

- *To substitute Multiple Inheritance* – Multiple inheritance is disallowed in Java. If a class has to derive properties from more than one class, it can have a inner class which can extend from another class.

Instance of an Inner class

- Similar to instance methods and variables, an inner class is associated with an instance of the outer class.

- Since inner class is associated with an instance, it cannot define any static members itself.

- An instance of the inner class can exist only in the context of an instance of the outer class and has direct access to methods and fields of its enclosing instance.

- To instantiate an inner class, first the outer class should be instantiated and then the inner class object is created within the outer object.

```java
public class OuterClass
{
        class InnerClass { }
        public static void main(String[ ] arg)
        {
                OuterClass outer = new OuterClass();
                OuterClass.InnerClass inner =
                                outer.new InnerClass();

        }
}
```

Example of Inner class

```java
class OuterClass {
    private int outer_x = 100;
    void test(){
    InnerClass inner=new InnerClass();
    inner.display();
    }
    class InnerClass {
        int inner_y = 10; // y is local to inner
        private void display(){
            System.out.println("display : outer_x = " +outer_x);
        }
    }
    void show(){
        System.out.println(inner_y);  //error, y not know here!
    }
}
```

Create an instance of InnerClass to access its members

Question time

Please try to limit the questions to the topics discussed during the session. Thank you.