# OO In Java

## Unit 3

# Unit 3 Objectives



- What is OOP?
- OOP Vs Structured Programming
- Classes & Objects
- Comparing Objects
- Access Specifers
- Constructors
- Static members
- Encapsulation
- Abstraction
- Inheritance
- Polymorphism
- Final modifier
- Inner Classes

# 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

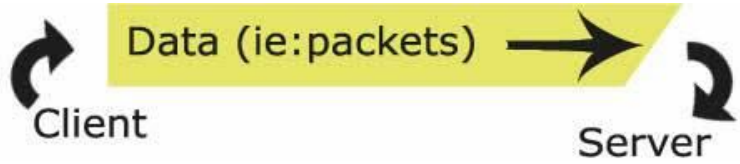



figure1: procedural

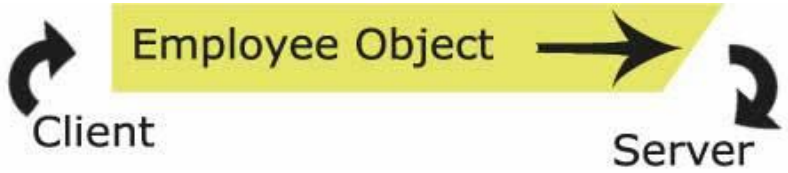
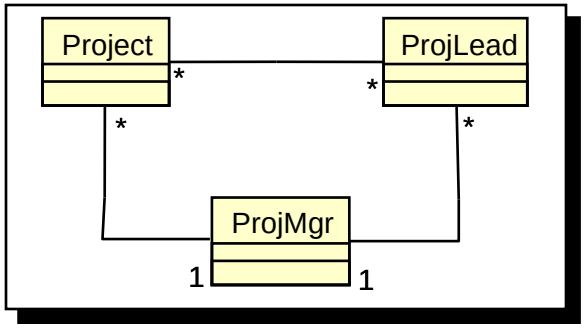


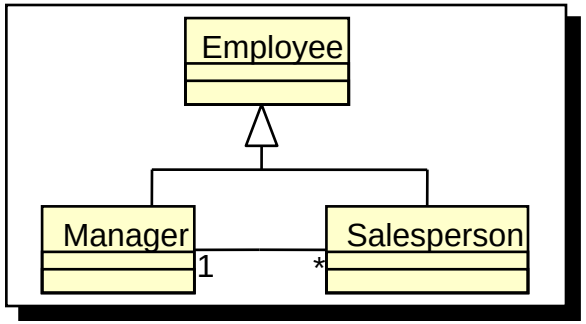

figure2: object-oriented

# Why choose the Object Oriented 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

# 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

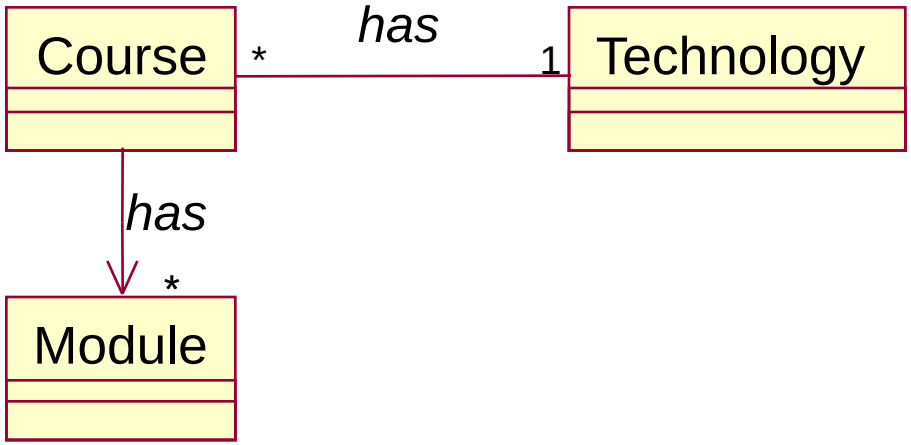
- Identify the different connections (relationships) between the above classes

# Identifying Relationships

- Trainer - Trainee
  - Trainer ‘HAS’ many Trainees
  - Every Trainee ‘HAS’ a Trainer

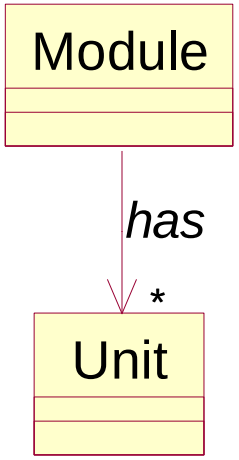
# Identifying Relationships

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

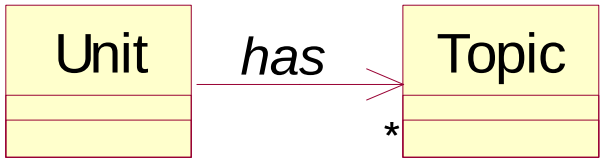


# Identifying Relationships

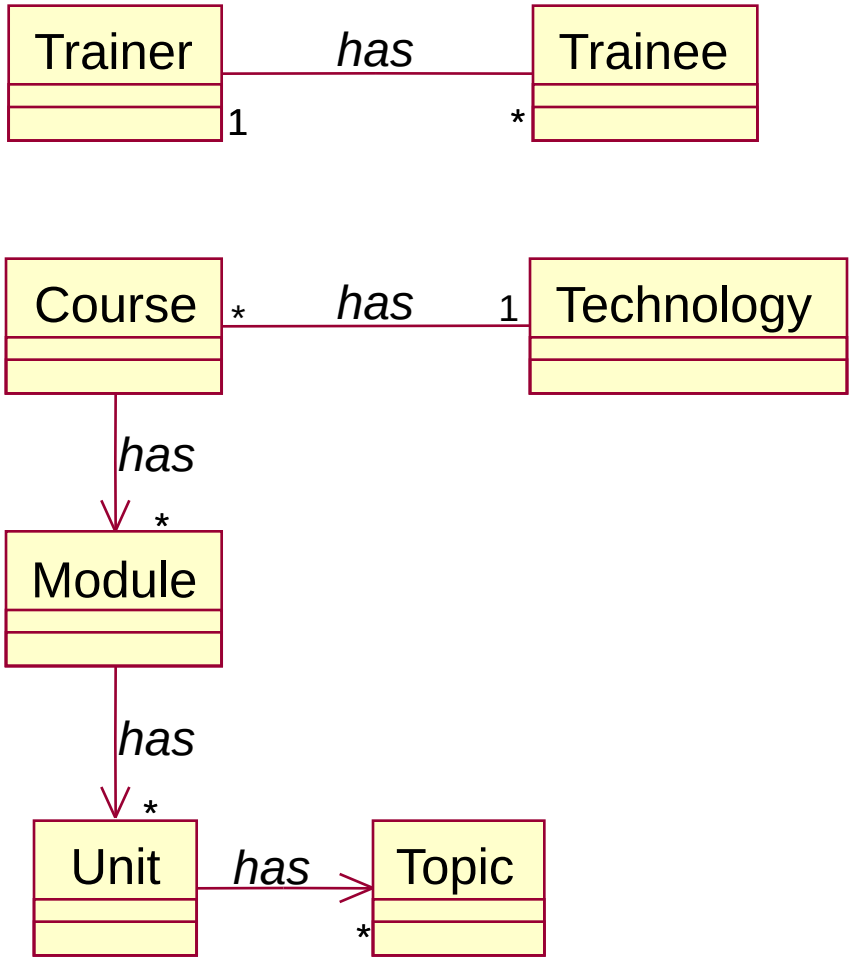
- Module – Unit
  - Module ‘HAS’ many Units




- Unit – Topic
  - Unit ‘HAS’ many Topics



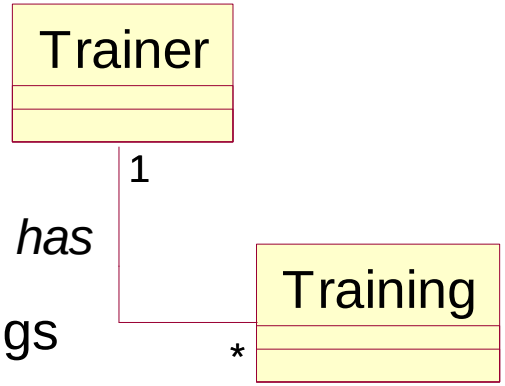
# The OO Model



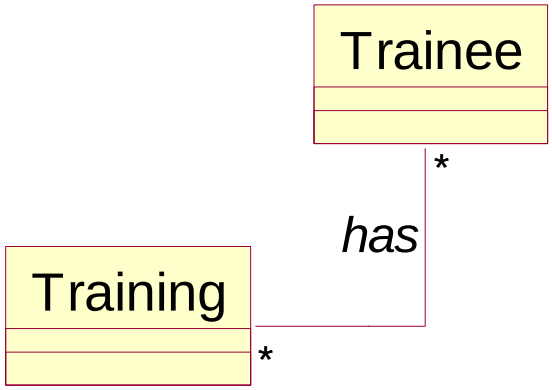

- How do you relate the Trainer & Trainee to the Course?

# Conceptual Entity

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

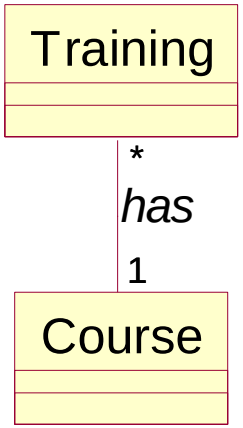



- Trainee – Training
  - A Trainee (HAS) attends many Trainings
  - A Training HAS a many Trainees

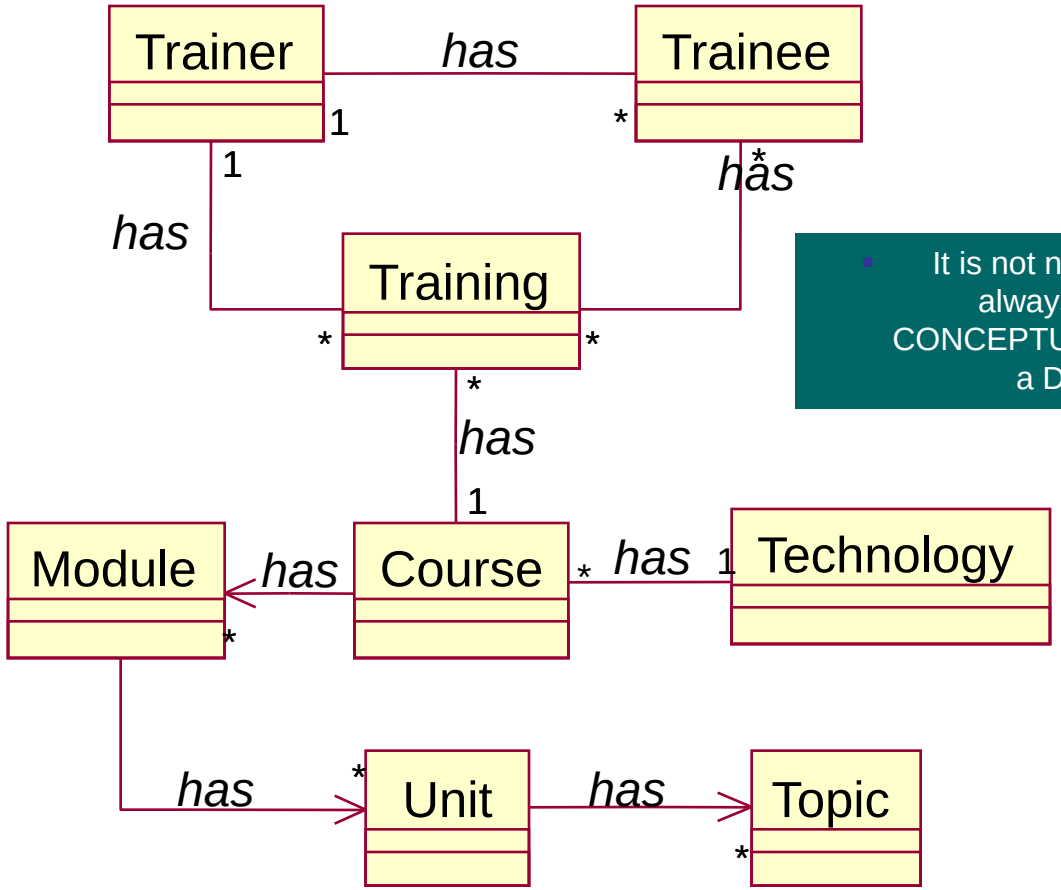


# Conceptual Entity

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



# 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.

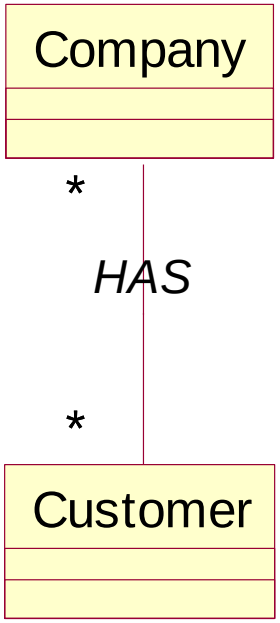
- Identify the different classes from the above problem statement
- Identify the different connections (relationships) between the above classes

# Identifying Classes

- Company
- Item
- Order
- Customer
- RegCustomer

# Identifying Relationships

- Company - Customer
  - Company ‘HAS’ many Customers
  - Customer ‘HAS’ many Companies

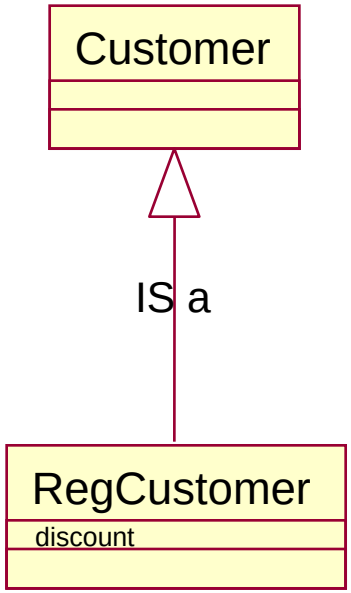


# Identifying Relationships

- Company - Item
  - Company HAS many Items

# Identifying Relationships

- Customer - RegCustomer
  - RegCustomer ‘IS’ a Customer

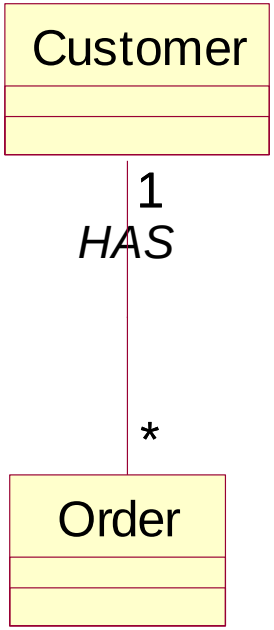


# Identifying Relationships

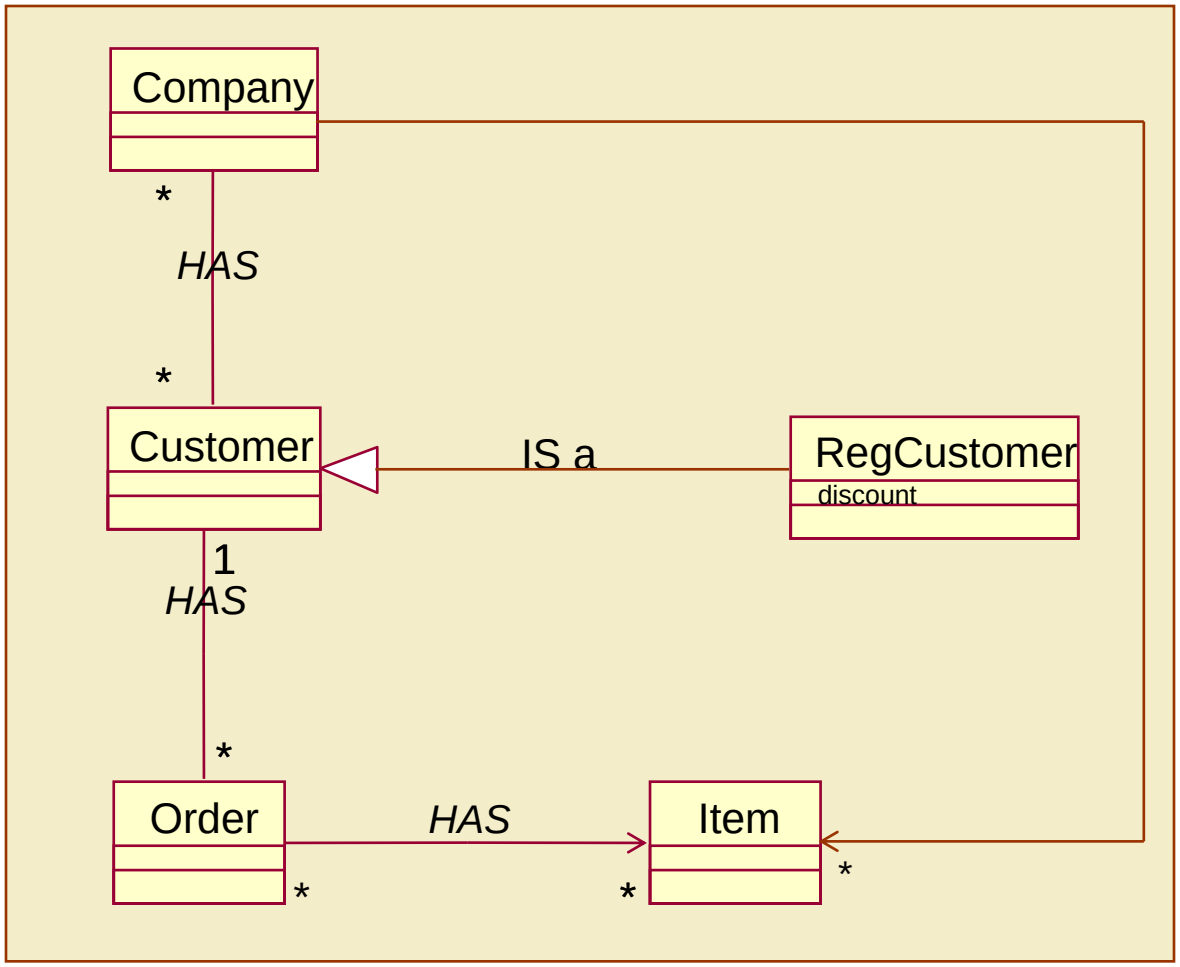
- Order - Item
  - Order HAS many Items

# Identifying Relationships

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



# The OO Model




**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**

# What is a Class?

- 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.

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

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

# Structure of a class

```
public class Employee
{
        private String employeeId;
        private String employeeName;

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

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

Variables

Constructor

Methods

# 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.

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

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

# Instantiating Classes



```
public class Shop
{
    public static 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
    - address : Address
  - Methods
    - Set methods and get methods for the data members
- The class Address has the following
  - Data
    - addr1 : string
    - addr2 : string
    - city : string
    - pin : int
  - Methods
    - set and get methods
- Write a class EmployeeDemo with a main and two methods
  - storeData() which takes the Employee object as an argument
    - Accepts user input for employee data and sets the data on the object
  - showData() which takes an Employee object as the argument and displays the data from the object
  - Create an instance of the Employee object and pass the same to the storeData() and showData()

# References Vs. Objects



```
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();
    }
}
```

# Comparing Objects

```
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 = p1;
                if(p1 == p2)
                        System.out.println("Objects are same");
                else
                        System.out.println("Objects are same");
        }
}
```

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

# 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

# Comparing Objects

```
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(10,20);
                Point p2 = new Point(3   0);
                if(p1.equals(p2))
                        System.out.println(“Objects are same”);
                else
                        System.out.println(“Objects are same”);
        }
}
```

In the example, p1.equals(p2), checks if the two objects are same by checking the contents by over-ridding the equals() method rather than just checking the references

# Exercise

- 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.

# 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**

# Why Constructors?

```
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?

```
public class Product
{
    int id;
    String name,category;
    float price,tax;
    public Product(int id , String name ,
                        String category , float price)
    {
        this.id = id;
        this.name = name;
        this.category = category;
        this.price = price;
        if (price>99 && category.equls("luxury"))
          tax = price*0.20;
```

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

# Constructor

- Constructor is a special method with the same name as the class.
- Constructors are called implicitly at the time of object creation.
- Constructor do not have a return type.
- Constructors can be overloaded.
- Every class has at least one constructor.
  - Either defined by programmer or the compiler provides a default constructor.

```
class Product
{
        public Product
        {
            //instantiate the
            variables
            ……..
            ……..
        }
}
```

- ***this*** and ***super*** are special keywords used in constructor and other methods
  - *this* refers to current object being constructed
  - *super* references the parent

See listing : **ConstructorDemo.java**

# Compiler and Constructors

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

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

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

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

A constructor defined by programmer. Compiler does not add anything

# Exercise

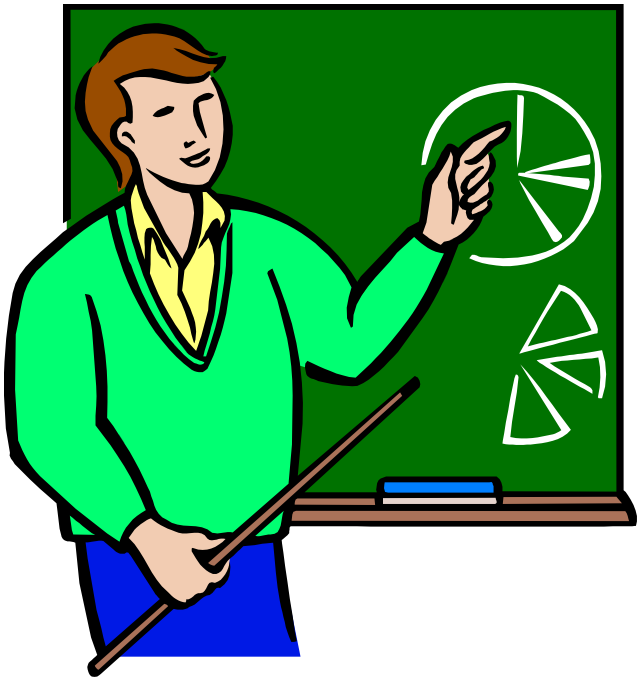
- Write a class Rectangle with data members, double length and double breadth. Write a parameterized constructor which takes the length and breadth as parameters and assigns it to the instance variables. Write a method area() which calculates the area and returns the same.
- Write a class InterestCalculator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

# Primary Object-oriented Principles

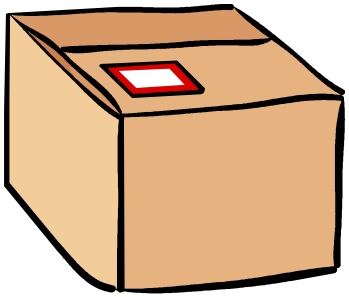
- Abstraction
- Encapsulation



# Abstraction

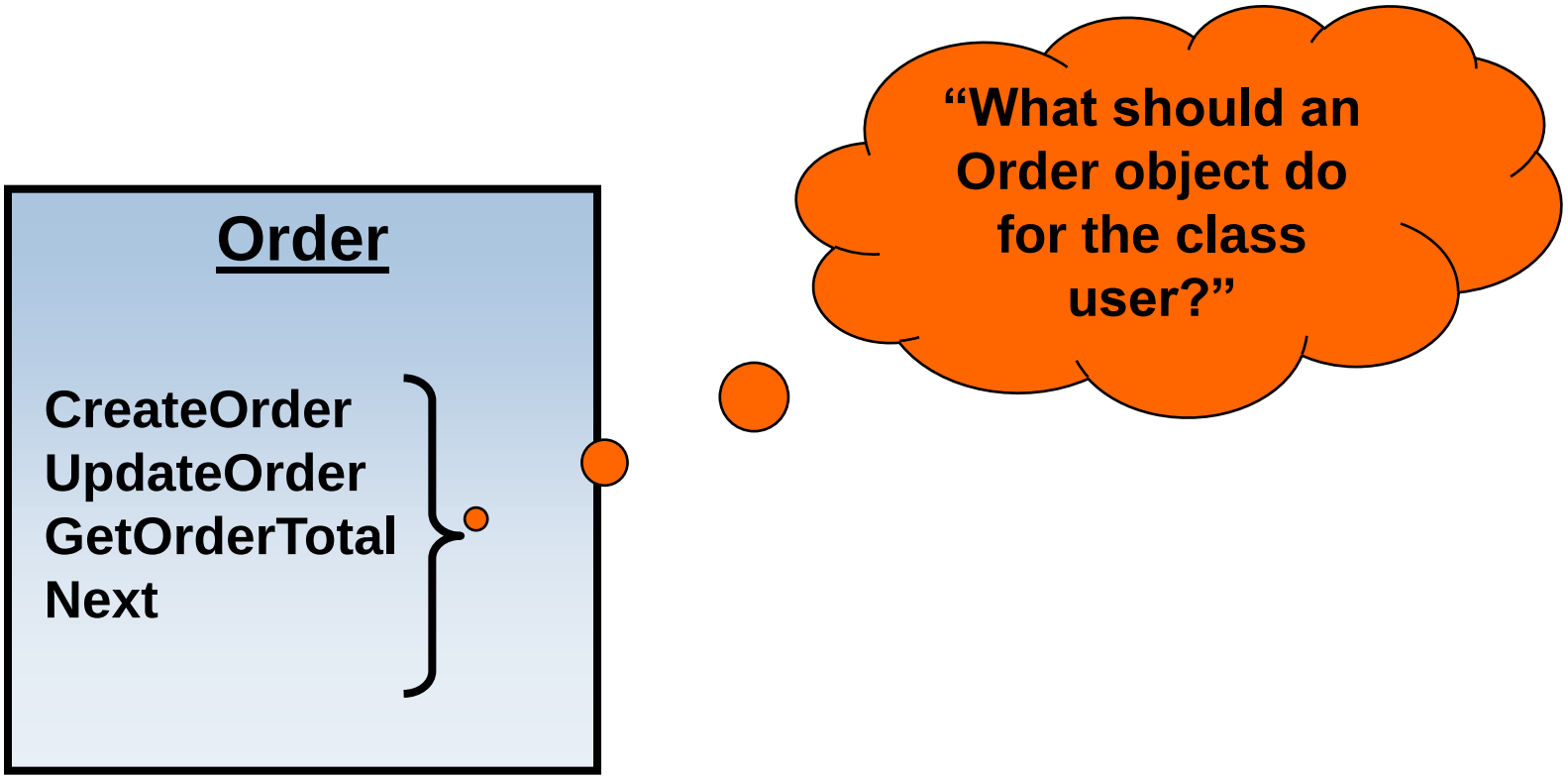


***Public View of an Object***



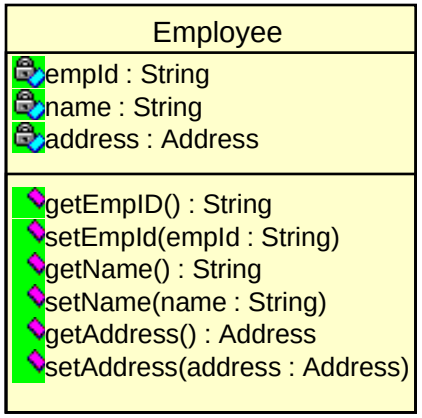
- *Abstraction* is used to minimize complexity for the class user
  - By allowing him focus on the essential characteristics
  - By hiding the details of implementation
- Simply put, abstraction is nothing but a process of ensuring that class users are not exposed to details which they do not need (or use).

# Abstraction - Example

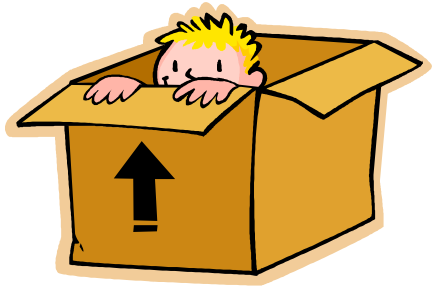


# Encapsulation

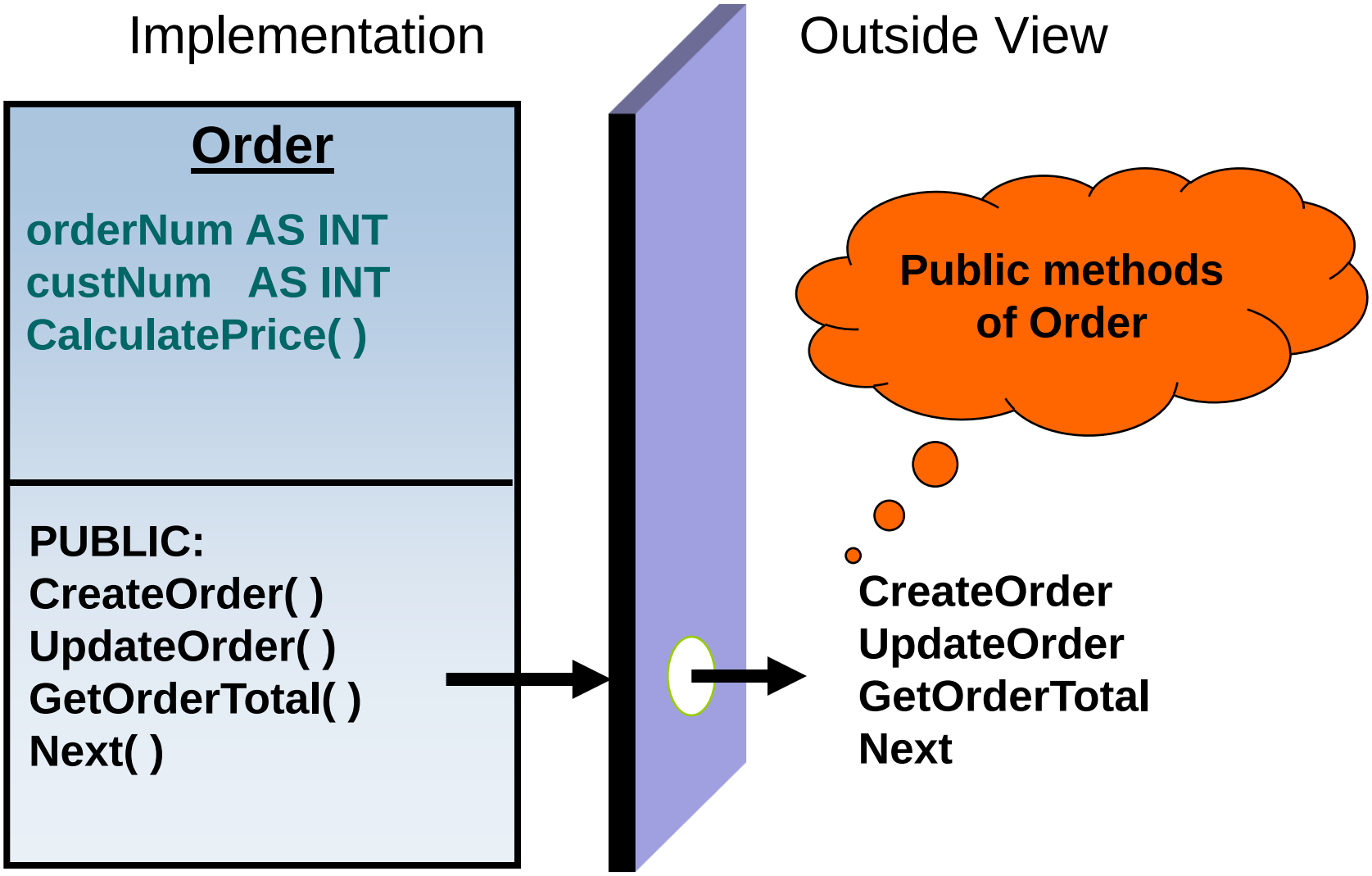
## *Hide Implementation Details*




- 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 - Example



# Identifying Abstraction & Encapsulation

**ABSTRACTIONS:**

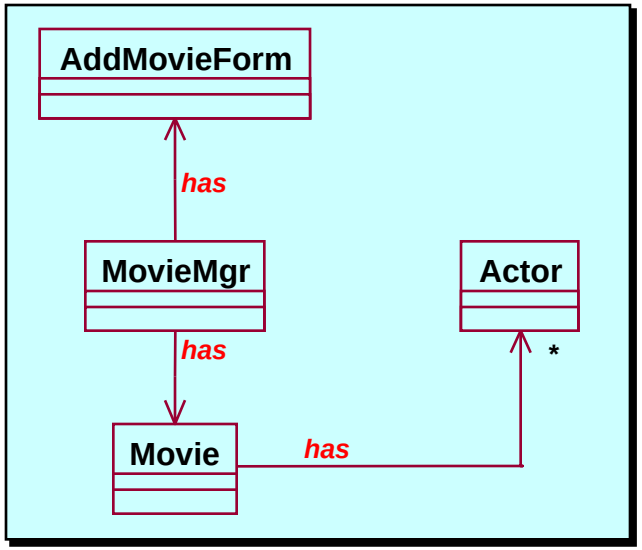
- What should the MovieMgr do?
- What are the responsibilities of the MovieMgr?

**MovieMgr**

**AcceptMovieInformation()**
**SetReleaseStatusOfMovie()**
**StoreMovieInformation()**
**AssignRolesToActor()**

**ENCAPSULATION:**

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




```
1
2  class MovieMgr
3  {
4      private AddMovieForm form;
5      private Movie movie;
6
7      public void acceptMovieInformation() { }
8      public void setReleaseStatusOfMovie() { }
9      public void storeMovieInformation() { }
10     public void assignRolesToActor(int index) { }
11 }
```

# Very Basic form of Abstraction

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

    void start()
    {
        ..........
    }
    void stop()
    {
        ..............
    }
}
```

Invisible

Visible

What should be exposed to the class user?

# 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

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

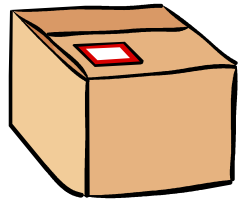
    public void start()
    {                                                  Visible
        ..........
    }
    public void stop()
    {
        ..............
    }
}
```

# If you are wondering…

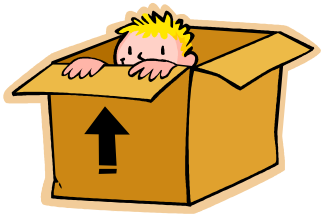
- If you are wondering what's the difference between Abstraction and encapsulation
- Encapsulation deals with 'what goes into' a class
- Abstraction deals with 'what is made visible' to the class user.

# Summary : Object-oriented Principles

- Abstraction
  - Break up complex problem
  - Focus on public view, commonalities



- Encapsulation
  - Hide implementation details
  - Package data and methods together



# 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.

```
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 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.

```
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;
        }
}
```

# Example of static methods

```
public 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;
       }
}
```

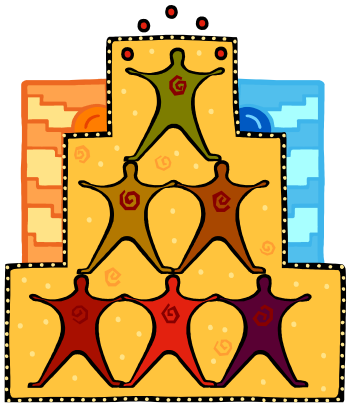
# Exercise

Write a class Printer which is implemented as a Singleton. The class has a method print() which takes a String as a parameter and prints the same on the console.

```
class Printer { }
class SingletonDemo
{
    public void client1()
    {
        Printer p = new Printer();
        p.print("String1");
    }
    public void client2()
    {
        Printer p = new Printer();
        p.print("String2");
    }
    public static void main(String[] args)
    {
        client1();
        client2();
    }
}
```

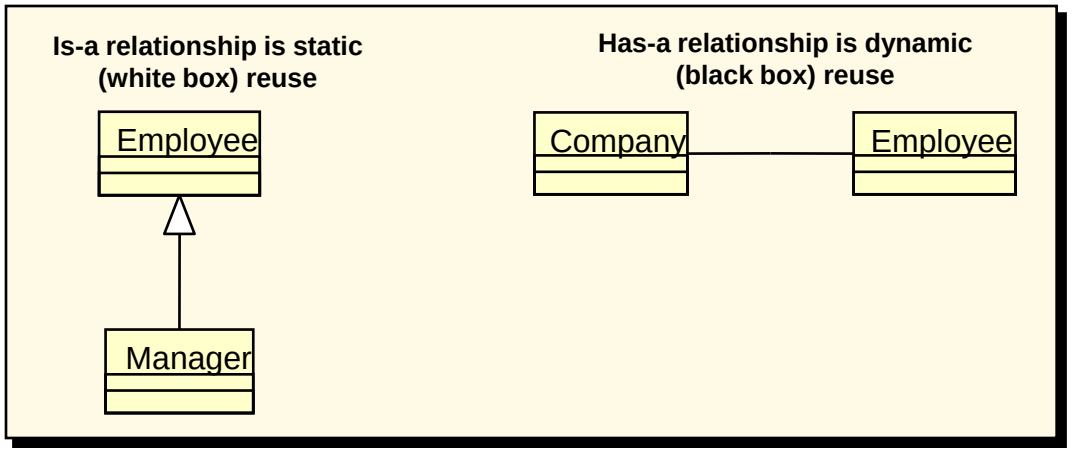
# Hierarchies

## *Object Relationships*



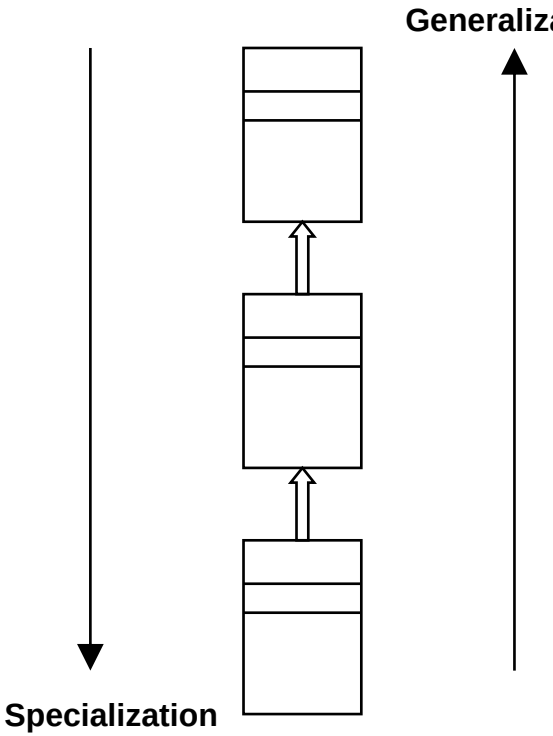
- Define relationships between objects
  - Objects defined in terms of other objects
  - Allows state and behavior to be shared and specialized as necessary
  - Encourages code reuse

- Two important hierarchy types:
  - Inheritance (Is-a)
  - Aggregation (Has-a)

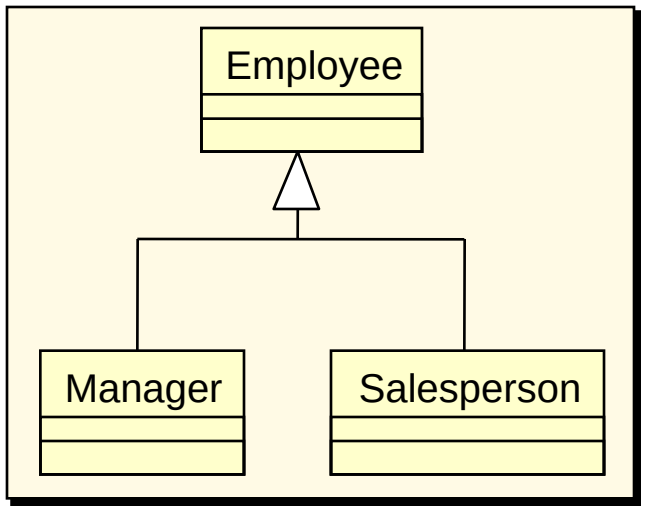


# Inheritance

- 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**




Mary




David

Joe

# Inheritance

- *Inheritance* is the process of obtaining common *attributes* and *behaviour* from another class.
- The parent class is called the *superclass* and the child class is called *subclass*.
- Allows hierarchical classification of objects
  - Similar to the biological classification of plants
- Subclass objects inherit all of the attributes of superclass objects
  - A deeply inherited subclass inherits all of the attributes from all super classes

# Why Inheritance?

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

    switchOnAC()
}
```

Redundant!!

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

    loadGoods()
    unloadGoods()
}
```

# Inheritance for Reuse

```
public class Vehicle
{
        Engine e;
        FuelTank tank;
        void pullFuelFromTank()
        void regulateEngineTemperature()
        void start()
        void stop()
}
```

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

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

# Inheritance and Access Specification

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

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

}
```

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

How would you achieve it?

Should be visible to everyone

# Inheritance and Access Specification

```
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.

# 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 Manger();//implicit
Manager mgr = new Manager();
mgr = (Manager) emp; // works  //explicit
emp = mgr; // works
```

# Type Casting of Objects

- A base class object cannot be typecast to a derived class reference.
  - eg

```
Employee emp  = new Employee();
Employee emp1 = new Employee();
Manager mgr = new Manager();
Manager mgr1 = new Employee(); //error
emp = mgr; // works
mgr = emp1; // error
```

See listing : **ObjectCastingDemo.java**

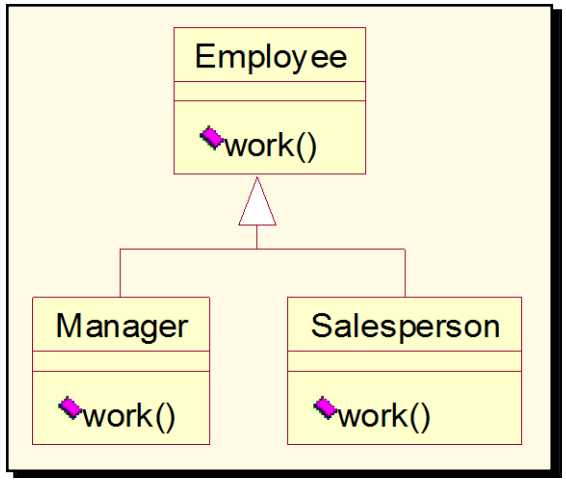
# Polymorphism

***One function, many implementations***

All employees do some work

Mary

David




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

# 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

# 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.

```
public class Addition
{

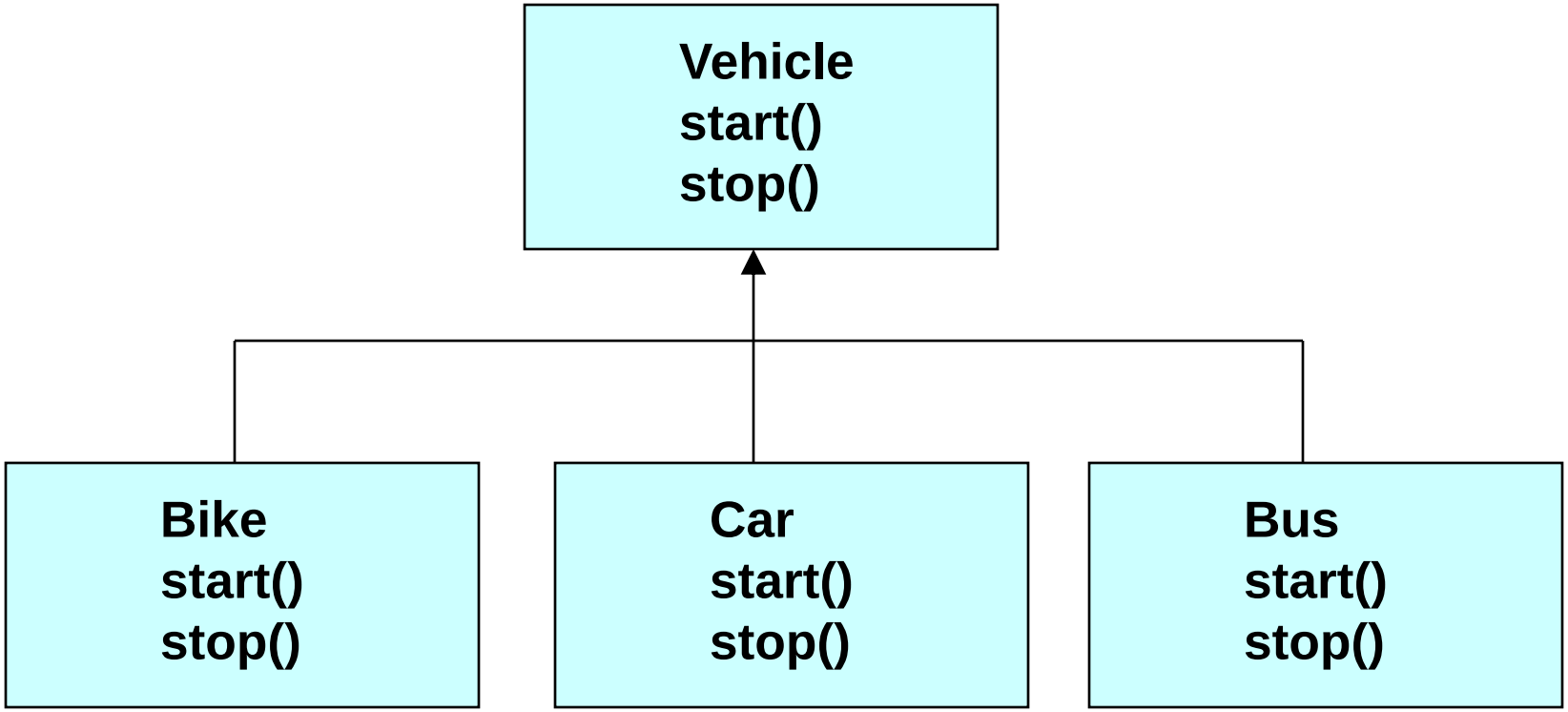
      public int add(double x ,
                     double y)
       {
             return x + y;
       }
       public int add(int x , int y)
       {
             return x + y;
       }
}
```

See listing : **OverLoadingDemo.java**

# Overloaded Constructor

```
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




See listing : **OverridingDemo1.java**

# 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 Vs 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.

# 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.

See listing : **FinalModifierDemo.java**

# 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.

```
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

```
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

# Static Inner Class

- Similar to a class methods and variables, a static inner class is associated with its outer class.
- The static nested class can access the various static members of the enclosing class.
- However, unlike non-static inner classes, it cannot access any instance variables in the enclosing class. This is an important difference between static nested classes and non-static inner classes.
- Formally, it can be said that an inner class is instance-scoped and a static nested class is class-scoped. The class scope of a static nested class makes it easier to use.

# Static Inner Class Example

```
class OuterClass
{
        private int outer_x = 100;
        private static int static_x = 200;

        class static InnerClass
        {
                private void display()
                {
                        System.out.println("display : outer_x = " +outer_x);
                        System.out.println("display : static_x = " +static_x);
                 }
        }
}
```

Error, cannot access non-static members

# Instantiating Static Inner Class

To instantiate a public static nested class, you do not need an instance of the outer class. You can simply use the class as is to create new instances.

```
class OuterClass
{
        public class static InnerClass
      {
            public void display()
            {
                  System.out.println("Inside inner display()");
            }
      }
}
Class OuterDemo
{
       public static void main(String[ ] args)
       {
            OuterClass.InnerClass inner = new OuterClass.InnerClass();
            inner.display();
       }
}
```

# Question time

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

Participants are encouraged to discuss other issues during the breaks.

Thank you.