UNIT – 1

Overview of Object Oriented concepts

Unit-01/Lecture-01

Introduction **[RGPV/DEC-2008(10)]**

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, and whose classes are all members of a hierarchy of classes united via inheritance relationships.

An object contains both data and methods that control the data. The data represents the state of the object. A class describes an object and they also form hierarchy to model real world system. The hierarchy is represented as inheritance and the classes can also be associated in different manners as per the requirement.

The objects are the real world entities that exist around us and the basic concepts like abstraction, encapsulation, inheritance, polymorphism all can be represented using UML.

So UML is powerful enough to represent all the concepts exists in object oriented analysis and design. UML diagrams are representation of object oriented concepts only. So before learning UML, it becomes important to understand OO concepts in details.

Following are some fundamental concepts of object oriented world:

Objects: Objects represent an entity and the basic building block.

Class: Class is the blue print of an object.

Abstraction: Abstraction represents the behavior of a real world entity.

Encapsulation: Encapsulation is the mechanism of binding the data together and hiding them from outside world.

Inheritance: Inheritance is the mechanism of making new classes from existing one.

Polymorphism: It defines the mechanism to exist in different forms.

Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Each object represents some entity of interest in the system being modeled, and is characterised by its class, its state (data elements), and its behavior. Various models can be created to show the static structure, dynamic behavior, and run-time deployment of these collaborating objects. There are a number of different notations for representing these models, such as the Unified Modeling Language (UML).

Object-oriented analysis (OOA) applies object-modelling techniques to analyze the functional requirements for a system. Object-oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on what the system does, OOD on how the system does it.

we dont take any liability for the notes correctness. http://www.rgpvonline.com

Object-oriented systems

An object-oriented system is composed of objects. The behavior of the system results from the collaboration of those objects. Collaboration between objects involves them sending messages to each other. Sending a message differs from calling a function in that when a target object receives a message, it itself decides what function to carry out to service that message. The same message may be implemented by many different functions, the one selected depending on the state of the target object.

The implementation of "message sending" varies depending on the architecture of the system being modeled, and the location of the objects being communicated with.

Object-oriented analysis

Object-oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt during object-oriented design (OOD). Analysis is done before the Design.

The sources for the analysis can be a written requirements statement, a formal vision document, interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object-oriented analysis is a description of what the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock-up. The purpose of object oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer defined requirements.

Object-oriented design

Object-oriented design (OOD) transforms the conceptual model produced in object-oriented analysis to take account of the constraints imposed by the chosen architecture and any non-functional – technological or environmental – constraints, such as transaction throughput, response time, run-time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of how the system is to be built.

What is UML?

Unified Modelling Language (UML) is the set of notations,models and diagrams used when developing object-oriented (OO) systems.

UML is the industry standard OO visual modelling language. The latest version is UML 1.4 and was formed from the coming together of three leading software methodologists; Booch, Jacobson and Rumbaugh.

UML allows the analyst ways of describing structure, behaviour of significant parts of system and their relationships.

Unified Modeling Language (UML) is a standardized general-purpose modeling language in the field of software engineering. The standard is managed, and was created by, the Object Management Group. UML includes a set of graphic notation techniques to create visual models of software-intensive systems.

The Unified Modeling Language is commonly used to visualize and construct systems which are software intensive. Because software has become much more complex in recent years, developers are finding it more challenging to build complex applications within short time periods. Even when they do, these software applications are often filled with bugs, and it can take programmers weeks to find and fix them. This is time that has been wasted, since an approach could have been used which would have reduced the number of bugs before the application was completed.

However, it should be emphasized that UML is not limited simply modeling software. It can also be used to build models for system engineering, business processes, and organization structures. A special language called Systems Modeling Language was designed to handle systems which were defined within UML 2.0. The Unified Modeling Language is important for a number of reasons. First, it has been used as a catalyst for the advancement of technologies which are model driven, and some of these include Model Driven Development and Model Driven Architecture.

Because an emphasis has been placed on the importance of graphics notation, UML is proficient in meeting this demand, and it can be used to represent behaviors, classes, and aggregation. While software developers were forced to deal with more rudimentary issues in the past, languages like UML have now allowed them to focus on the structure and design of their software programs. It should also be noted that UML models can be transformed into various other representations, often without a great deal of effort. One example of this is the ability to transform UML models into Java representations.

This transformation can be accomplished through a transformation language that is similar to QVT. Many of these languages may be supported by OMG. The Unified Modeling Language has a number of features and characteristics which separate it from other languages within the same category. Many of these attributes have allowed it to be useful for developers. In this article, I intend to show you many of these attributes, and you will then understand why the Unified Modeling Language is one of the most powerful languages in existence today.

Objects and classes: **[RGPV/DEC-2010(10)]**

Objects are composite data types. An *object* provides for the storage of multiple data values in a single unit. Each value is assigned a name which may then be used to reference it. Each element in an object is referred to as a *property*. Object properties can be seen as an unordered list of name value pairs contained within the container object.

Object comes in two flavors. There are system defined objects, which are predefined and come with the JavaScript parser or with the browser running the parser. And there are user defined objects, which the programmer creates.

Class a definition, or description, of how the object is supposed to be created, what it contains, and how it work

There are two important concepts to understand when talking about objects. These are the ideas of class and instance.
Creating objects is a two step process. First you must define a class of objects, then you use the object class by declaring instances of that class within the program. The object class is a definition, or description, of how the object is supposed to be created, what it contains, and how it works. The object instance is a composite data type, or object, created based on the rules set forth in the class definition.

instance: a composite data type, or object, created based on the rules set forth in the class definition

This break between class and instance is not new, it is just that before objects, all data classes were hard coded into the parser and you could just make use of them while creating variables that were instances of those classes. Someone, somewhere, had to write the code to define the integer data type as being a numeric value with no fractional component. Whenever you declare an integer variable, you make use of this definition to create, or instantiate, an integer. Fortunately for us, it all happens behind the scenes.

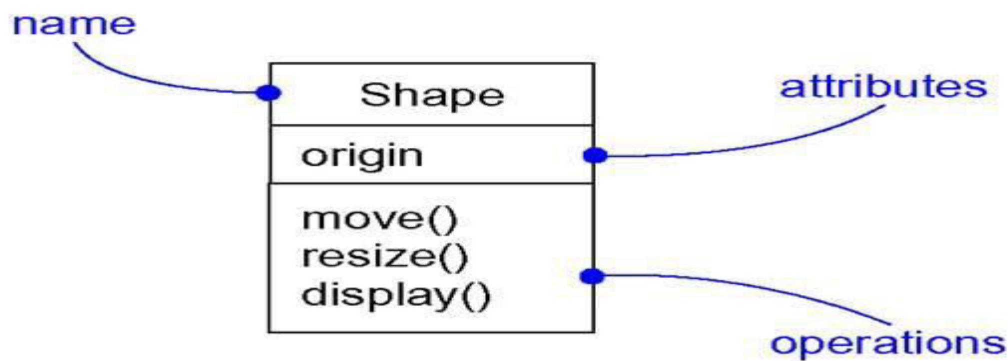
The point of object-based programming languages is that they give the user the ability to define their own data types that can be specifically tailored to the needs of the application. There are still system-defined data types and object classes, so you don't need to worry about defining commonly used types of variables, but you now can go beyond them.
Since objects are composite data types, they can contain more than one piece of data. In fact, the very point of the object is to bring together related data elements into a logical grouping. This grouping can contain not only data values, but also rules for processing those values. In an object, a data element is called a property, while the rules the object contains for processing those values are called methods. This makes objects very powerful because they can not only store data, but they can store the instructions on what to do with that data.

```
public class Student {
}
```

According to the sample given below we can say that the student object, named objectStudent, has created out of the Student class.

```
Student objectStudent = new Student();
```

Classes



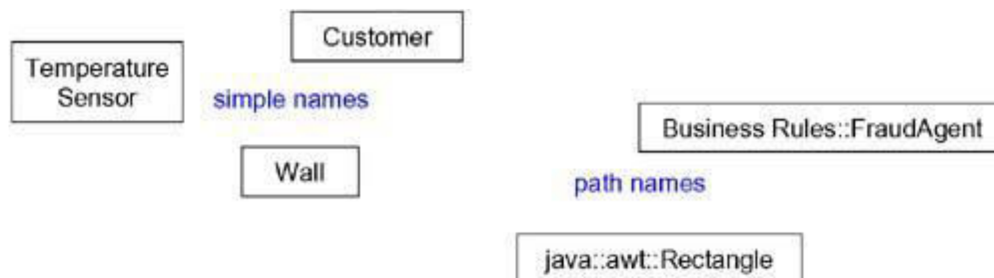
Terms and Concepts

A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics. Graphically, a class is rendered as a rectangle.

Names

A class name must be unique within its enclosing package,
Every class must have a name that distinguishes it from other classes. A *name* is a textual string. That name alone is known as a *simple name;* a *path name* is the class name prefixed by the name of the package in which that class lives. A class may be drawn showing only its name,

Simple and Path Names



Note

A class name may be text consisting of any number of letters, numbers, and certain punctuation marks (except for marks such as the colon, which is used to separate a class name and the name of its enclosing package) and may continue over several lines. In practice, class names are short nouns or noun phrases drawn from the vocabulary of the system you are modeling. Typically, you capitalize the first letter of every word in a

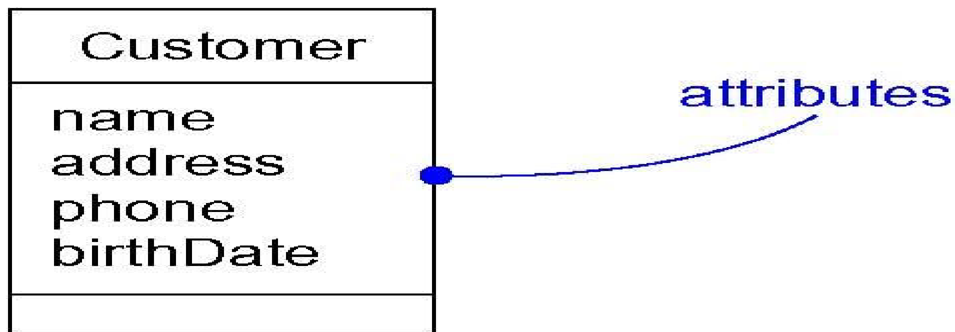
class name, as in Customeror TemperatureSensor.

Attributes

Attributes are related to the semantics of aggregation.

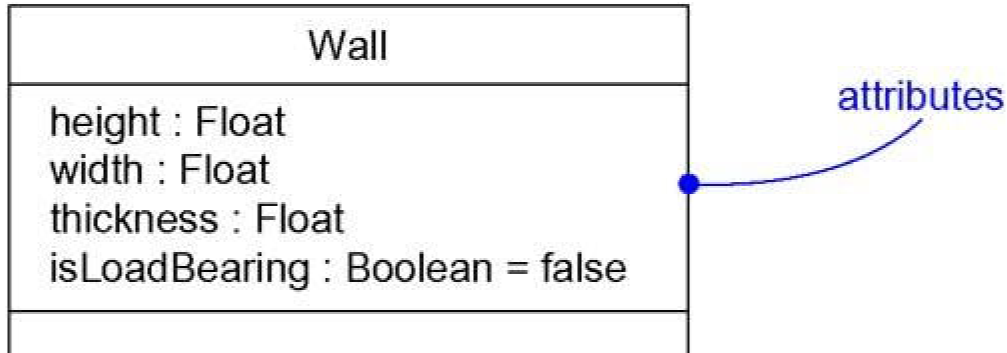
An *attribute* is a named property of a class that describes a range of values that instances of the property may hold. A class may have any number of attributes or no attributes at all. An attribute represents some property of the thing you are modeling that is shared by all objects of that class. For example, every wall has a height, width, and thickness; you might model your customers in such a way that each has a name, address, phone number, and date of birth. An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass. At a given moment, an object of a class will have specific values for every one of its class's attributes. Graphically, attributes are listed in a compartment just below the class name. Attributes may be drawn showing only their names.

Attributes



Note

An attribute name may be text, just like a class name. In practice, an attribute name is a short noun or noun phrase that represents some property of its enclosing class. Typically, you capitalize the first letter of every word in an attribute name except the first letter, as in nameor loadBearing.



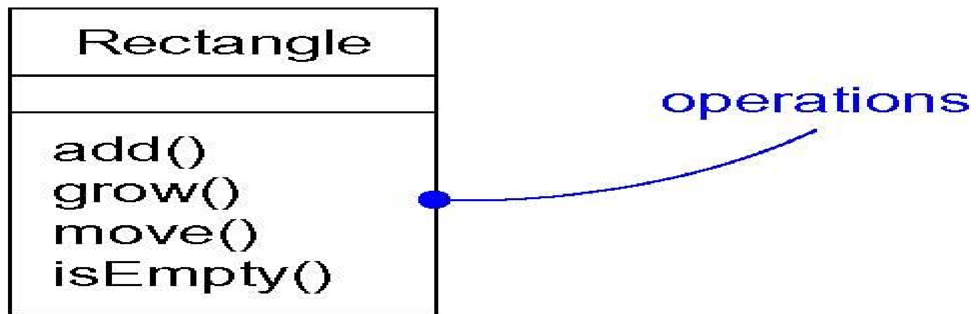
Attributes and Their Classes

S.NO	RGPV QUESTIONS	Year	Marks
1.	DESCRIBE THE MECHANISM OF ACCESING DATA MEMBERS and member functions in the following cases: a) inside the main program. b)inside the member function of the same class. c)inside a member function of another class.	Dec 2013	7
2.	What are different forms of inheritance? Give examples of each.	Dec 2013	7
3.	In what order are the class constructors called when a derived class objects are created.	Dec 2013	7
4	How is polymorphism achieved at compile time and run time.	Dec 2013	7
5	What is object oriented analysis and design?	Dec, 2008	10
6	What is object orientation in software development?	Dec, 2010	10

Operations **[RGPV/JUNE-2006(10)]**

An *operation* is the implementation of a service that can be requested from any object of the class to affect behavior. In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class. A class may have any number of operations or no operations at all. For example, in a windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties. Often (but not always), invoking an operation on an object changes the object's data or state. Graphically, operations are listed in a compartment just below the class attributes. Operations may be drawn showing only their names.

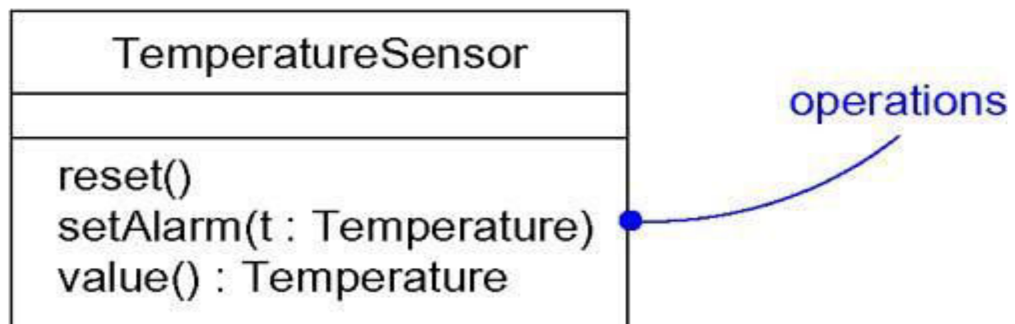
Operations



Note

An operation name may be text, just like a class name. In practice, an operation name is a short verb or verb phrase that represents some behavior of its enclosing class. Typically, you capitalize the first letter of every word in an operation name except the first letter, as in moveor isEmpty.

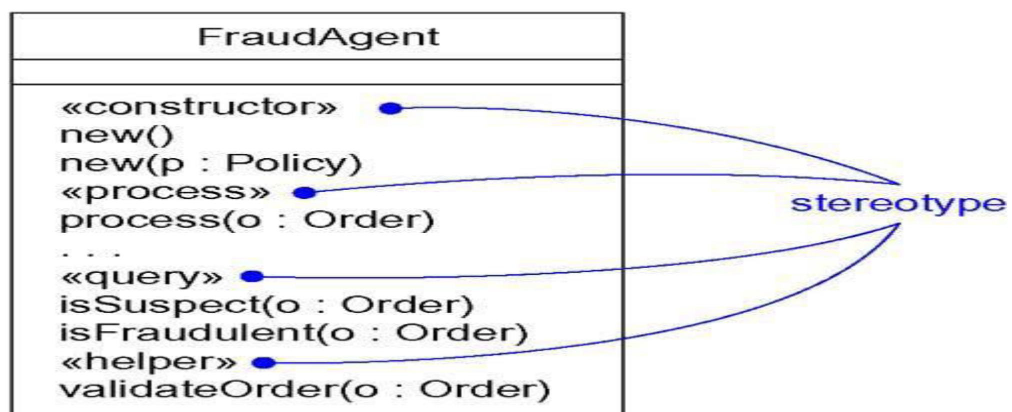
Operations and Their Signatures



Organizing Attributes and Operations

When drawing a class, you don't have to show every attribute and every operation at once. In fact, in most cases, you can't (there are too many of them to put in one figure) and you probably shouldn't (only a subset of these attributes and operations are likely to be relevant to a specific view). For these reasons, you can elide a class, meaning that you can choose to show only some or none of a class's attributes and operations. An empty compartment doesn't necessarily mean there are no attributes or operations, just that you didn't choose to show them. You can explicitly specify that there are more attributes or properties than shown by ending each list with an ellipsis ("...").

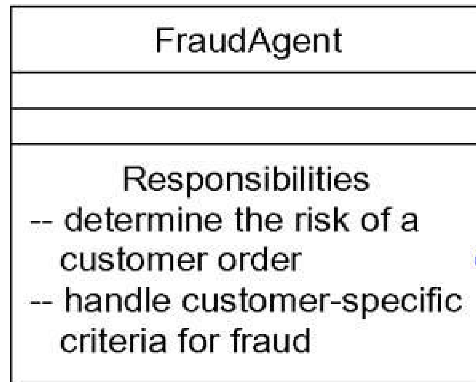
Stereotypes for Class Features



Responsibilities

A *responsibility* is a contract or an obligation of a class. When you create a class, you are making a statement that all objects of that class have the same kind of state and the same kind of behavior. At a more abstract level, these corresponding attributes and operations are just the features by which the class's responsibilities are carried out. A Wall class is responsible for knowing about height, width, and thickness; a FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent; a TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

When you model classes, a good starting point is to specify the responsibilities of the things in your vocabulary. Techniques like CRC cards and use case-based analysis are especially helpful here. A class may have any number of responsibilities, although, in practice, every well-structured class has at least one responsibility and at most just a handful. As you refine your models, you will translate these responsibilities into a set of attributes and operations that best fulfill the class's responsibilities.



Responsibilities

Note

Responsibilities are just free-form text. In practice, a single responsibility is written as a phrase, a sentence, or (at most) a short paragraph.

S.NO	RGPV QUESTIONS	Year	Marks
1.	Describe the method for identifying classes and objects	June, 2006	10

Abstraction [RGPV /DEC-2010(5)]

One job of an OO developer is to take a problem domain, and from it deduce which classes will be needed, and what instance/variables go in each class.

- Generally easy to identify the lowest-level classes, but we often want to make use of inheritance!

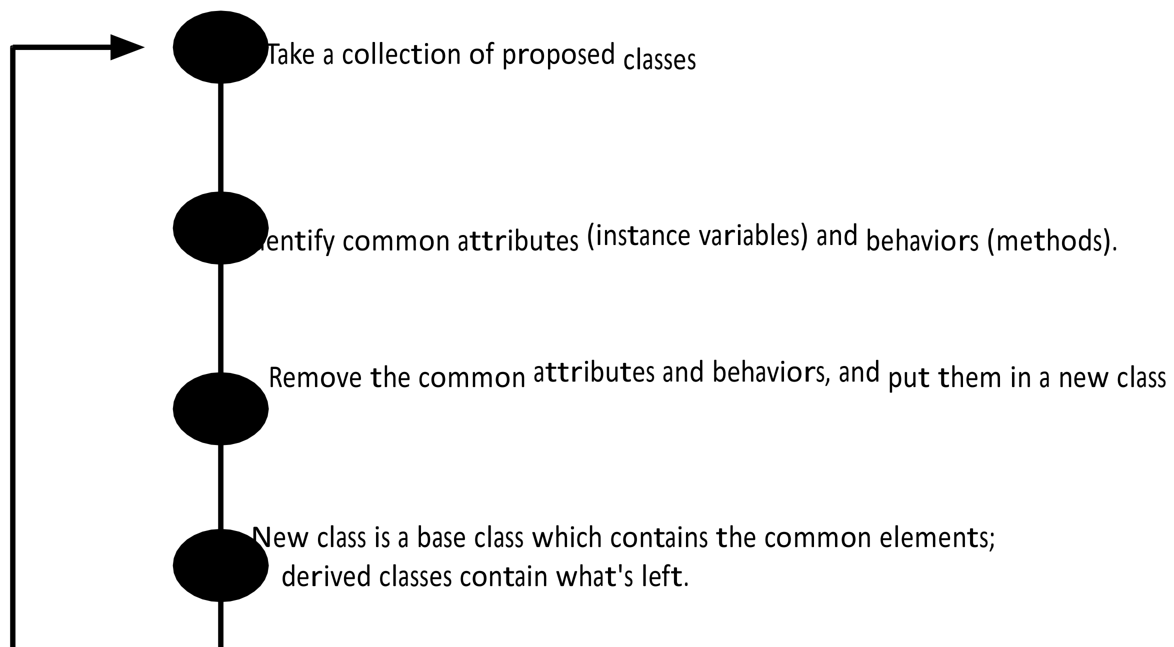
Abstraction is the technique of deciding...

– What classes do we need?

– How do we organize our class hierarchy?

– Which variables/methods do we put in the superclasses, and which do we put in the subclasses?

Basic approach to abstraction

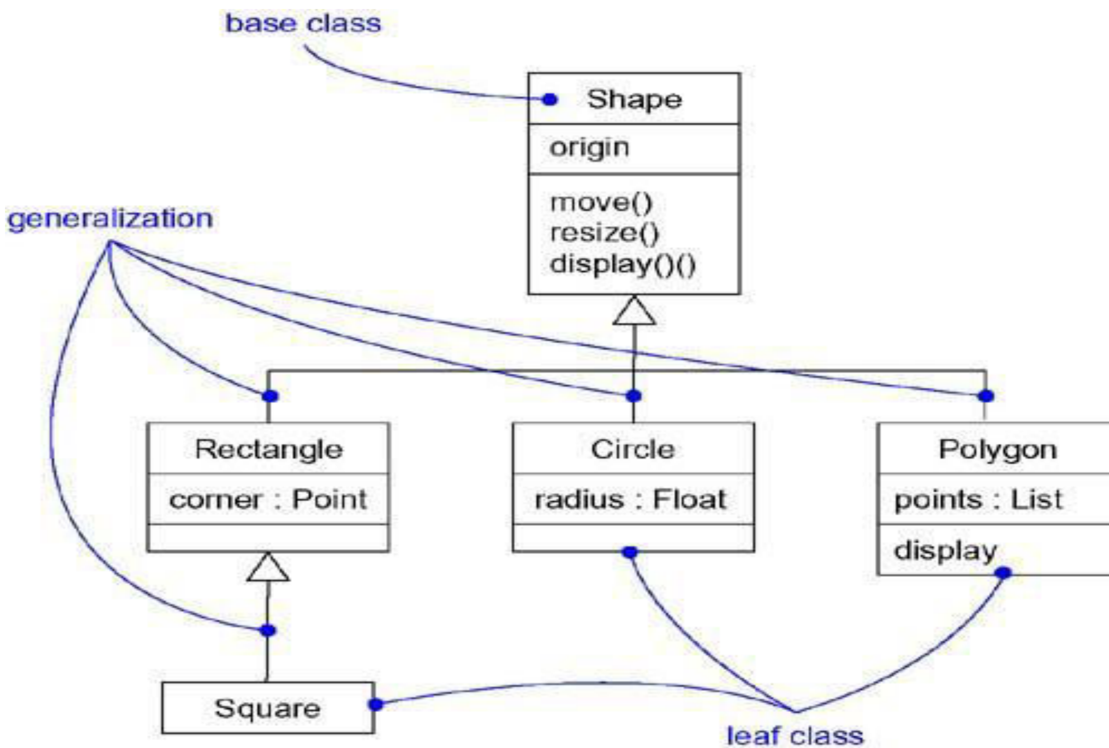


Generalization

generalization is a relationship between a general thing (called the superclass or parent)and a more specific kind of that thing (called the subclass or child). Generalization is sometimes called an "is-a-kind-of" relationship: one thing (like the class BayWindow) is-a-kind-of a more general thing (for example, the class Window). Generalization means that objects of the child may be used anywhere the parent may appear, but not the reverse. In other words, generalization means that the child is substitutable for the parent. A child inherits the properties of its parents, especially their attributes and operations. Often • but not always •

the child has attributes and operations in addition to those found in its parents. An operation of a child that has the same signature as an operation in a parent overrides the operation of the parent; this is known as polymorphism. Graphically, generalization is rendered as a solid directed line with a large open arrowhead, pointing to the parent. Use generalizations when you want to show parent/child relationships.

Generalization



A class may have zero, one, or more parents. A class that has no parents and one or more children is called a root class or a base class. A class that has no children is called a leaf class. A class that has exactly one parent is said to use single inheritance; a class with more than one parent is said to use multiple inheritance.

Most often, you will use generalizations among classes and interfaces to show inheritance relationships. In the UML, you can also create generalizations among other things• most notably, packages.

Note

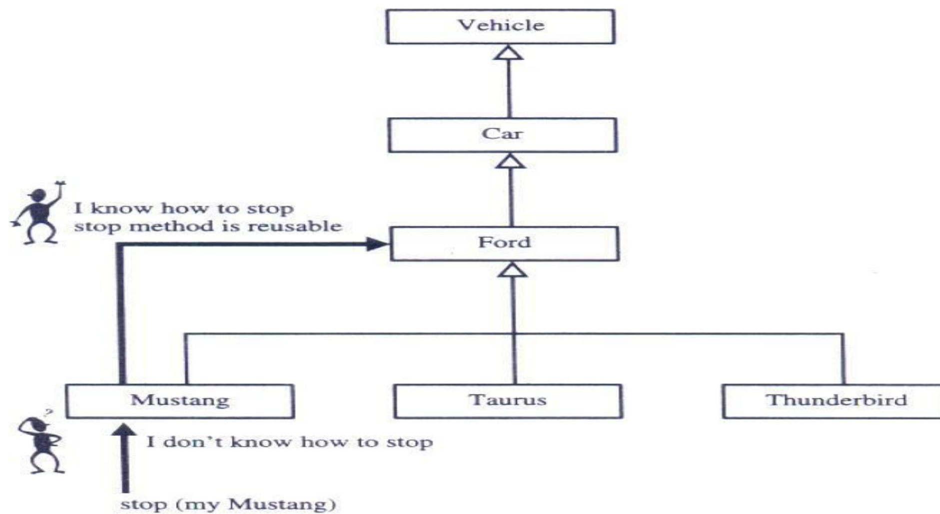
A generalization can have a name, although names are rarely needed unless you have a model with many generalizations and you need to refer to or discriminate among generalizations.

S.NO	RGPV QUESTIONS	Year	Marks
1	What is data abstraction? How can we implement it in object oriented languages	Dec,2010	5

Inheritance **[RGPV /DEC-2008(5)]**

Inheritance is the property of object-oriented systems that allows objects to be built from other objects. Inheritance allows explicitly taking advantage of the commonality of objects when constructing new classes. Inheritance is a relationship between classes where one class is the parent class of another (derived) class. The parent class also is known as the *base class* or *super class.* Inheritance provides programming by extension as opposed to programming by reinvention The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class's methods, a class also inherits the behaviors and attributes of all of its super classes.

For example, the Car class defines the general behavior of cars. The Ford class inherits the general behavior from the Car class and adds behavior specific to Fords. It is not necessary to redefine the behavior of the car class; this is inherited. Another level down, the Mustang class inherits the behavior of cars from the Car class and the even more specific behavior of Fords from the Ford class. The Mustang class then adds behavior unique to Mustangs. Assume that all Fords use the same braking system. In that case, the *stop* method would be defined in class Ford (and not in Mustang class), since it is a behavior shared by all objects of class Ford. When you step on the brake pedal of a Mustang, you send a *stop* message to the Mustang object. However, the *stop* method is not defined in the Mustang class, so the hierarchy is searched until a *stop* method is found. The *stop* method is found in the Ford class, a super class of the Mustang class, and it is invoked. In a similar way, the Mustang class can inherit behaviors from the Car and the Vehicle classes.



INHERITANCE ALLOWS REUSABLITY

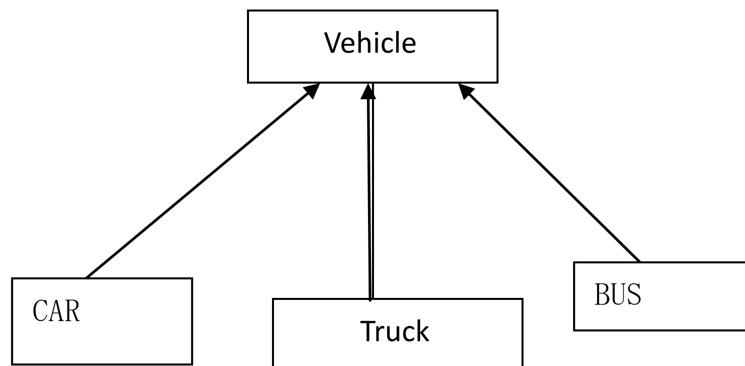
we dont take any liability for the notes correctness.

http://www.rgpvonline.com

Inheritance for Specialization

One common reason to use inheritance is to create specializations of existing classes. In specialization, the derived class has data or behavior aspects that are not part of the base class. For example, Square is a Rectangle. Square class is specialization of Rectangle class. Similarly, Circle is an Ellipse. Here also, Circle class is specialization of Ellipse class. Another example, a BankAccount class might have data members such as accountNumber, customerName and balance. An InterestBearingAccount class might inherit
BankAccount and then add data member interestRate and interestAccrued along with behavior for calculating interest earned.
Another form of specialization occurs when a base class specifies that it has a particular behavior but does not actually implement the behavior. Each non- abstract, concrete class which inherits from that abstract class must provide an implementation of that behavior. This providing of actual behavior by a subclass is sometimes known as implementation or reification.
For example, there is a class Shape having operation area(). The operation area() cannot be implemented unless we have concrete class. So, Shape class is abstract class. Rectangle is a Shape. Now, Rectangle is a concrete class, which can implement the operation area().

Inheritance for Generalization



Generalization is reverse of specialization. For instance, a "fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple is-a fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant. Another example: Vehicle is a generalization of Car, Truck, Bus etc. as shown in Figure 2.18. Car, Truck, Bus etc. share some properties such as “number of wheels”, speed, capacity etc. these common properties are abstracted out and put into another class say Vehicle, which comes higher in the hierarchy.

2.2.6.3 Inheritance for Extension

In this case, inheritance extends the existing class functionalities by adding new operations in the derived class. It can be distinguished from generalization that the later must override at least one method from the base and the functionality is tied to that of the base

class. Extension simply adds new methods to those of the base class and functionality is less strongly tied to the existing methods of the base class.

For example, StringSet class inherits from Set class, which specializes for holding strinvalues. Such a class might provide additional methods for string related operations – for instance - search by prefix, which returns a subset of all the elements of the set that begin with a certain string value. These operations are meaningful to the derived class but are not particularly relevant to the base class.

S.NO	RGPV QUESTIONS	Year	Marks
1.	What does inheritance means in object oriented programming?	Dec,2008	5

Unit-01/Lecture-05

Inheritance for Restriction **[RGPV /DEC-2009(10)]**

In this case, the derived class does not implement the functionality, which a base class has. In other words, inheritance for restriction occurs when the behaviour of the derived class is smaller or more restrictive than the behaviour of the base class.

For example, an existing class library provides a double-ended queue (deque). Elements can be added or removed from either end of the deque, but the programmer wishes to write a stack class, enforcing the property that elements can be added or removed from only one end of the stack. Here, the programmer can make the Stack class a derived class of the existing Deque class and can modify or override the undesired methods so that they produce an error message if used.

2.2.6.5 Inheritance for Overriding

When a class replaces the implementation of a method that it has inherited is called overriding. Overriding introduces a complication: which version of the method does an instance of the inherited class use the one that is part of its own class, or the one from the parent (base) class. The answer varies between programming languages, and some languages provide the ability to indicate that a particular behaviour is not to be overridden.

2.2.6.6. Constraints of inheritance-based design

When using inheritance extensively in designing a program, one should be aware of certain constraints that it imposes. For example, consider a class Person that contains a person's name, address, phone number, age, gender, and race. We can define a subclass of Person called Student that contains the person's grade point average and classes taken, and another subclass of Person called Employee that contains the person's job title, employer, and salary.

In defining this inheritance hierarchy we have already defined certain restrictions, not all of which are desirable:

- Singleness: Using single inheritance, a subclass can inherit from only one super class. Continuing the example given above, Person can be either a Student or an Employee, but not both. Using multiple inheritance partially solves this problem, as a Student Employee class can be defined that inherits from both Student and Employee. However, it can still inherit from each super class only once; this scheme does not support cases in which a student has two jobs or attends two institutions.
- Static: the inheritance hierarchy of an object is fixed at instantiation when the object's type is selected and does not change with time. For example, the inheritance graph does not allow a Student object to become a Employee object while retaining the state of its Person super class.

  (Although similar behaviour can be achieved with the decorator pattern.)

Some have criticized inheritance, contending that it locks developers into their original design standards.

- Visibility: whenever client code has access to an object, it generally has access to all the object's superclass data. Even if the superclass has not been declared public, the client can still cast the object to its superclass type. For example, there is no way to give a function a pointer to a Student's grade point average and transcript without also giving that function access to all of the personal data stored in the student's Person superclass.

2.2.6.7 Roles and inheritance:

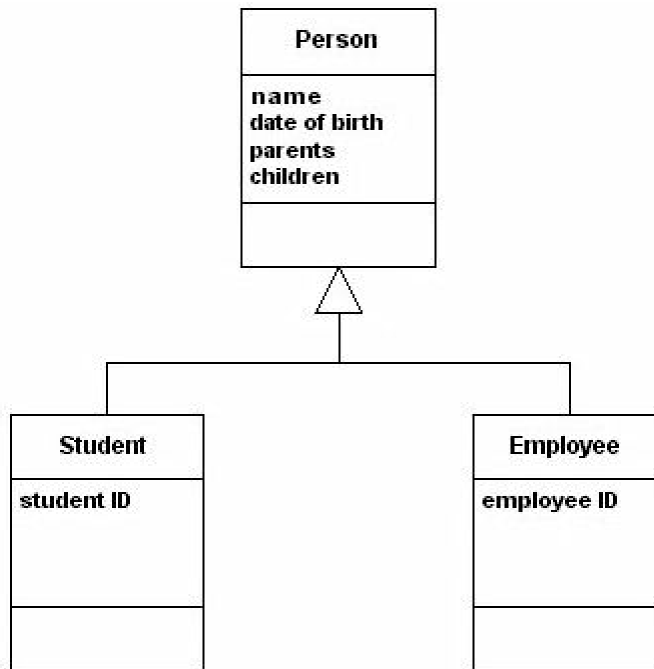
One consequence of separation of roles and super classes is that compile- time and run-time aspects of the object system are cleanly separated. Inheritance is then clearly a compile-time construct. Inheritance does influence the structure of many objects at run-time, but the different kinds of structure that can be used are already fixed at compile-time.
To model the example of Person as an employee with this method, the modelling ensures that a Person class can only contain operations or data that are common to every Person instance regardless of where they are used. This would prevent use of a Job member in a Person class, because every person does not have a job, or at least it is not known that the Person class is only used to model Person instances that have a job. Instead, object-oriented design would consider some subset of all person objects to be in an "employee" role. The job information would be associated only to objects that have the employee role. Object-oriented design would also model the "job" as a role, since a job can be restricted in time, and therefore is not a stable basis for modelling a class. The corresponding stable concept is either "Workplace" or just "Work" depending on which concept is meant. Thus, from object- oriented design point of view, there would be a "Person" class and a "Workplace" class, which are related by a many-to-many association "works- in", such that an instance of a Person is in employee role, when he works-in a job, where a job is a role of his work place in the situation when the employee works in it.

Note that in this approach, all classes that are produced by this design process are part of the same domain, that is, they describe things clearly using just one terminology. This is often not true for other approaches.

The difference between roles and classes is especially difficult to understand if referential transparency is assumed, because roles are types of references and classes are types of the referred-to objects.

In this example, a Student is a type of Person. Likewise, a Employee is a type of Person. Both Student and Employee inherit all the attributes and methods of Person. Student has a locally defined student ID attribute. Employee has a locally defined employee ID attribute.

So, if you would look at a Student object, you would see attributes of name, date of birth, parents, children, and student ID.



S.NO	RGPV QUESTIONS	Year	Marks
1.	What are the advantages of code reusability? What is containership? How does it differ from inheritance	Dec,2009	10

UNIT 1/LECTURE 6

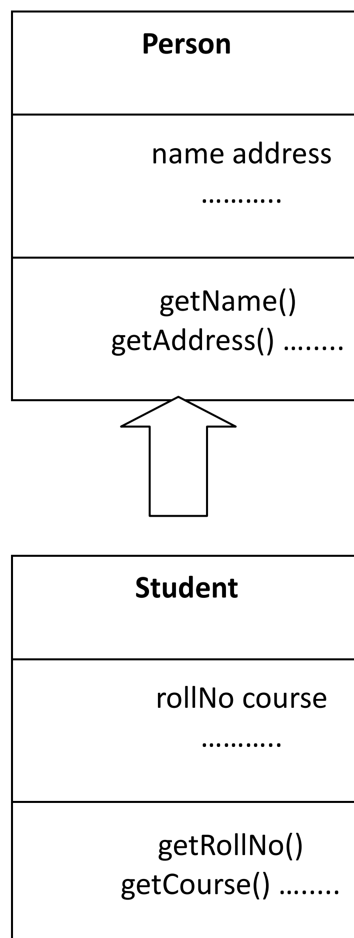
Types of Inheritance **[RGPV /DEC-2008(8),DEC-2010(8)]**

There are many ways a derived class inherits properties from the base class.

Following are the types of inheritance:

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multipath Inheritance
- Hybrid Inheritance

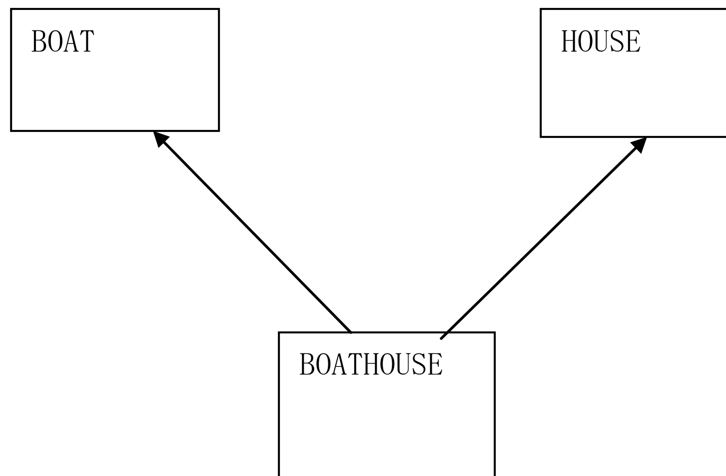
Single Inheritance



When a (derived) class inherits properties (data and operations) from a single base class, it is called as single inheritance. For example, Student class inherits properties from Person class.

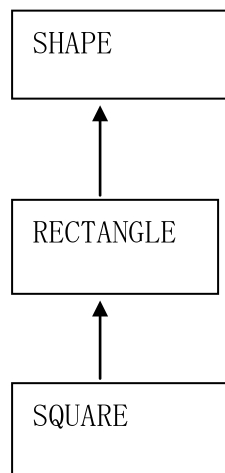
Multiple Inheritance

When a (derived) class inherits properties (data and operations) from more than one base class, it is called as multiple inheritance. For example, BoatHouse class inherit properties from both Boat class and House class.



Multilevel Inheritance

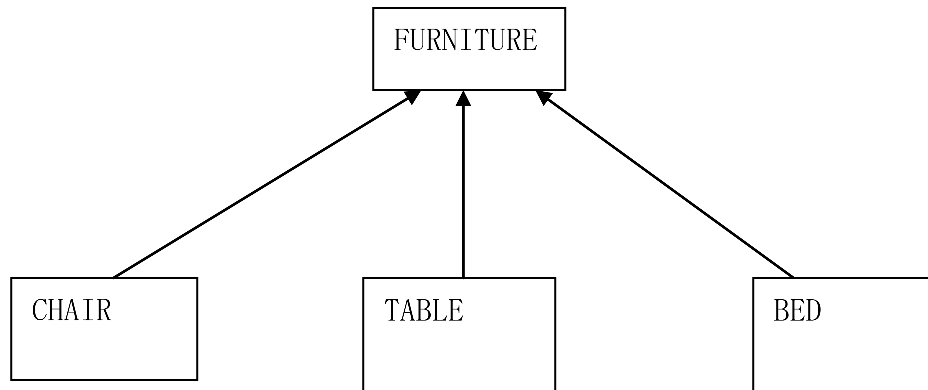
When a (derived) class inherits properties (data and operations) from another derived class, it is called as multilevel inheritance. For example, Rectangle class inherits properties from Shape class and Square inherits properties from Rectangle class.



Hierarchical Inheritance

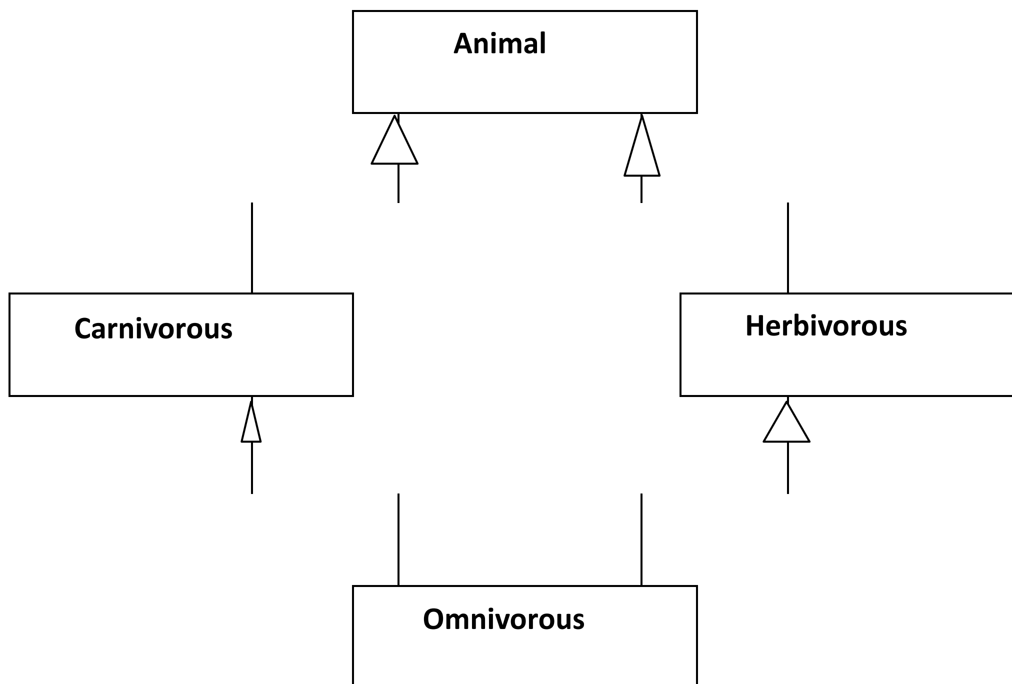
When more than one (derived) class inherits properties (data and operations) from a single base class, it is called as hierarchical inheritance. For example, Chair class, Table class and

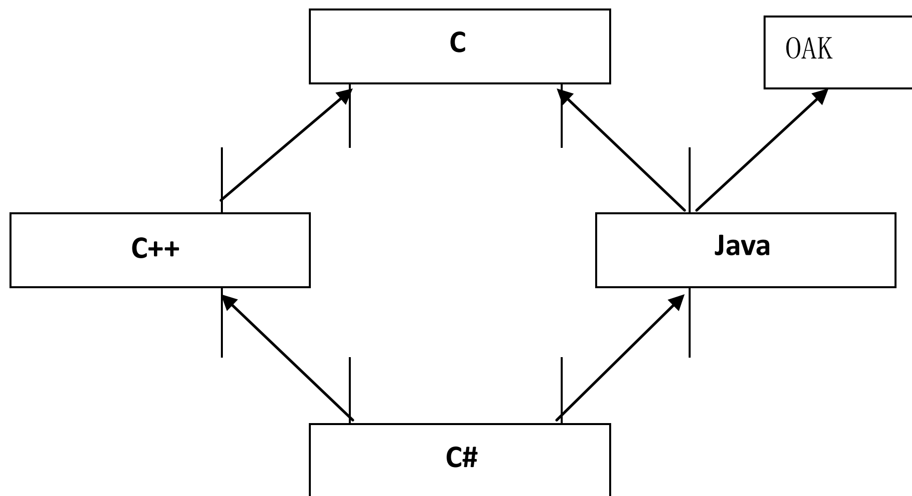
Bed class all inherit properties from Furniture class.



Multipath Inheritance

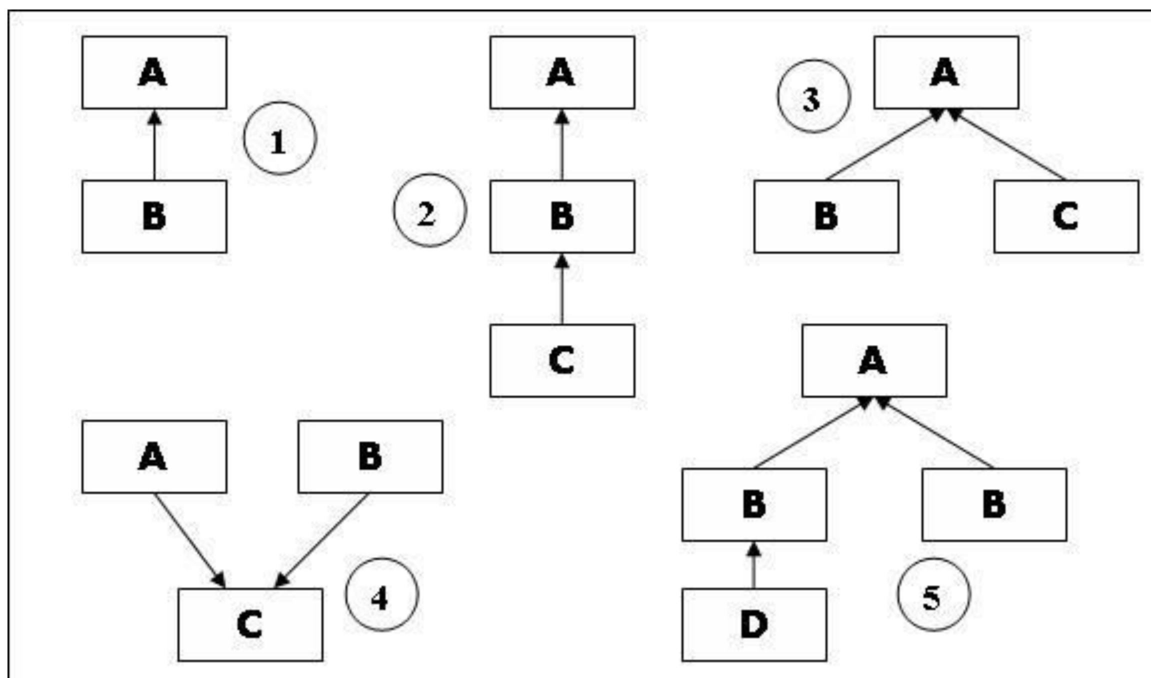
When more than one inheritance paths are available between two classes in the inheritance hierarchy, it is called as multipath inheritance. For example, Carnivorous and Herbivorous class inherit properties from Animal class. Omnivorous class inherits properties from Carnivorous and Herbivorous classes. So, there are two alternative paths available from Animal class to Omnivorous class.





Hybrid Inheritance

Mixture of single, multiple, hierarchical and multilevel inheritance forms hybrid inheritance



S.NO	RGPV QUESTION	YEAR	MARKS
1.	What are the different forms of inheritance ? give an example for each	Dec 2010,Dec,2008	8
2	Dicuss in brief multiple inheritance and disinheritance	Dec,2008	4

UNIT 1/LECTURE 7

DYNAMIC INHERITANCE

Dynamic inheritance allows objects to change and evolve over time. Since base classes provide properties and attributes for objects, changing base classes changes the properties and attributes of a class. A previous example was a Windows object changing into an icon and then back again, which involves changing a base class between a Windows class and an Icon class. More specifically, *dynamic inheritance* refers to the ability to add, delete, or change parents from objects (or classes) at run time.
In object-oriented programming languages, variables can be declared to hold or reference objects of a particular class. For example, a variable declared to reference a motor vehicle is capable of referencing a car or a truck or any subclass of motor vehicle.

MULTIPLE INHERITANCE

Some object-oriented systems permit a class to inherit its state (attributes) and behaviors from more than one super class. This kind of inheritance is referred to as *multiple inheritance.* For example, a utility vehicle inherits attributes from both the Car and Truck classes.

Multiple inheritance can pose some difficulties. For example, several distinct parent classes can declare a member within a multiple inheritance hierarchy. This then can become an issue of choice, particularly when several super classes define the same method. It also is more difficult to understand programs written in multiple inheritance systems.

One way of achieving the benefits of multiple inheritance in a language with single inheritance is to inherit from the most appropriate class and then add an object of another class as an attribute.

Encapsulation **[RGPV /DEC-2010(5)]**

Encapsulation is one of the loosely defined OOAD concepts. The term is known in software development for many years but I can't find any reliable origin. Encapsulation was mentioned in the article describing **abstraction** mechanisms in programming language CLU in the context of **hiding details of implementation**.

CLU restricted access to the implementation by allowing using only (public) cluster operations, i.e. public **interface**. It promoted design practices where abstractions are used to define and simplify the connections between system modules and to **encapsulate** implementation decisions that are likely to change.

If we look up the English word **encapsulate** in a dictionary, we will find two meanings: (1) to encase or become enclosed in a capsule (2) to express in a brief summary, epitomise. Both of these meanings of encapsulation seem appropriate in the context of OOAD.

Let's assume that the definition of encapsulation in OOAD is something like:

Encapsulation is a development technique which includes

- creating new data types (**classes**) by combining both information (structure) and behaviors, and
- restricting access to implementation details.

Encapsulation is very close or similar to the abstraction concept. The difference is probably in "direction" - encapsulation is more about hiding (encapsulating) implementation details while abstraction is about finding and exposing public interfaces. The two concepts are supported by access control.

Access control allows both to hide implementation (**implementation hiding** or **information hiding**) and to expose public interface of a class.

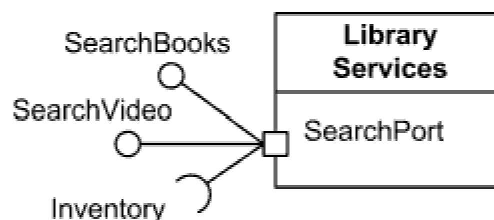
Encapsulation in UML

UML specifications provide no definition of **encapsulation** but use it loosely in several contexts.

For example, in UML 1.4 **object** is defined as an entity with a well defined boundary and identity that **encapsulates** state (attributes and relationships) and behavior (operations, methods, and state machines). Elements in peer packages are **encapsulated** and are not a priori **visible** to each other.

In UML 2.4 and 2.5 a component represents a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment, and also a Component is encapsulated and ... as a result, Components and subsystems can be flexibly reused and replaced by connecting ("wiring") them together.

Encapsulated classifier in UML 2.4 and 2.5 is a structured classifier isolated from its environment (encapsulated ?) by using ports. Each port specifies a distinct interaction point between classifier and its environment.

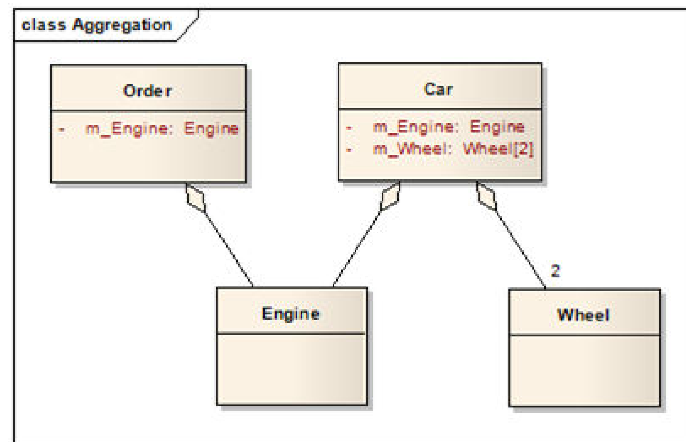


Library Services is classifier encapsulated through Search Port

UML 2.4 specification also used term **completely encapsulated** without providing any explanation. It was removed in UML 2.5.

Aggregation **[RGPV /JUNE-2008(10)]**

A plain association between two classes represents a structural relationship between peers, meaning that both classes are conceptually at the same level, no one more important than the other. Sometimes, you will want to model a "whole/part" relationship, in which one class represents a larger thing (the "whole"), which consists of smaller things (the "parts"). This kind of relationship is called aggregation, which represents a "has-a" relationship, meaning that an object of the whole has objects of the part. Aggregation is really just a special kind of association and is specified by adorning a plain association with an open diamond at the whole end.



Note

The meaning of this simple form of aggregation is entirely conceptual. The open diamond distinguishes the "whole" from the "part," no more, no less. This means that simple aggregation does not change the meaning of navigation across the association between the whole and its parts, nor does it link the lifetimes of the whole and its parts.

S.NO	RGPV QUESTION	YEAR	MARKS
1.	What is data encapsulation? How can we implement in any object oriented language	Dec,2010	5
2.	Discuss among association,aggregation, inheritance and relationship among classes,quoting suitable examples	June 2008	10

UNIT 1/LECTURE 8

AGGREGATIONS AND OB.JECT CONTAINMENT

All objects, except the most basic ones, are composed of and may contain other objects. For example, a spreadsheet is an object composed of cells, and cells are objects that may contain text, mathematical formulas, video, and so forth. Breaking down objects into the objects from which they are composed is decomposition. This is possible because an object's attributes need not be simple data fields; attributes
can reference other objects. Since each object has an identity, one object can refer to other objects. This is known as *aggregation,* where an attribute can be an object itself. For instance, a car object is an aggregation of engine, seat, wheels, and other objects (see Figure2.9).
A Car object is an aggregation of other objects such as engine, seat, *and wheel objects.*

Abstraction classes:

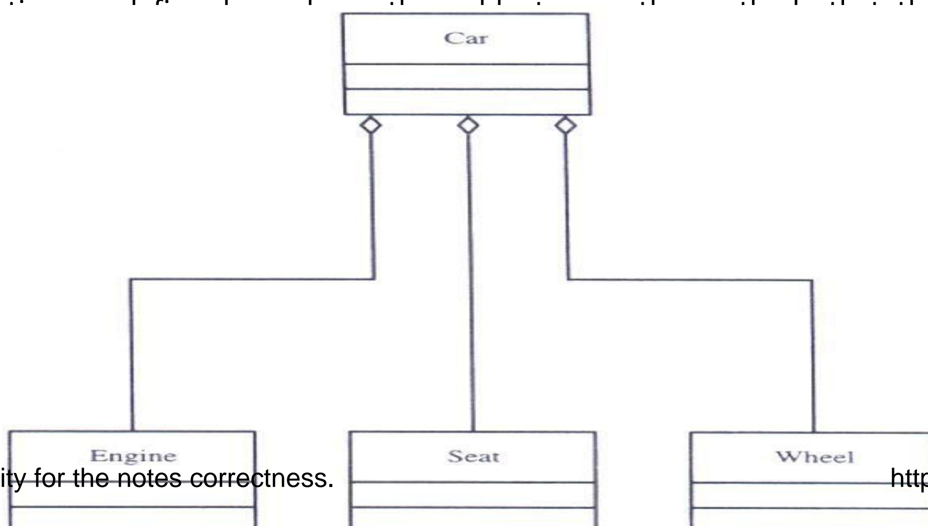
Classes with no instances are called *abstract classes*. An abstract class is written with the expectation that its subclasses will add to its structure and behavior, usually by completing the implementation of its (typically) incomplete methods. In fact, in Smalltalk a developer may force a subclass to redefine the method introduced in an abstract class by using the method **subclassResponsibility** to implement a body for the abstract class's method. If the subclass fails to redefine it, then invoking the method results in an execution error. C++ similarly allows the developer to assert that an abstract class's method cannot be involced direaly by initializing its declaration to zero. Such a method is called a *pure virtual function*, and the language prohibits the creation of instances whose class exports such functions.

Standard protocols are often represented by *abstract classes*.

An abstract class never has instances, only its subclasses have instances. The roots of class hierarchies are usually abstract classes_while the leaf classes are never abstract. Abstract classes usually do not define any instance variables. However, they define methods in terms of a few undefined methods that must be implemented by the subclasses. For example, class Collection is abstract, and defines a number of methods, including select:,collect:, and inject:into:, in terms of an iteration method, do:. Subclasses of Collection, such as Array,
Se ... erited
frc ... e the
us

A ... stract
cla ... data
re ... stract
cla ... any
re

Cr ... use a
nic ... ng in



we dont take any liability for the notes correctness. http://www.rgpvonline.com

Smalltalk makes it easier to discover the important abstractions. A Smalltalk programmer always tries to create new classes by making them be subclasses of existing ones, since this is less work than creating a class from scratch. This often results in a class hierarchy whose top-most class is concrete. The top of a large class hierarchy should almost always be an abstract class, so the experienced programmer will then try to reorganize the class hierarchy and find the abstract class hidden in the concrete class. The result will be a new abstract class that can be reused many times in the future.

An example of a Smalltalk class that needs to be reorganized is View, which defines a user-interface object that controls a region of the screen. View has 27 subclasses in the standard image, but is concrete. A careful examination reveals a number of assumptions made in View that most of its subclasses do not use. The most important is that each view will have subviews. In fact, most subclasses of View implement views that can never have subviews. Quite a bit of code in View deals with adding and positioning subviews, making it very difficult for the beginning programmer to understand the key abstractions that View represents. The solution is simple: split View into two classes, one (View) of which is the abstract superclass and the other (ViewWithSubviews) of which is a concrete subclass that implements the ability to have subviews. The result is much easier to understand and to reuse.

Inheritance vs. decomposition

Since inheritance is so powerful, it is often overused. Frequently a class is made a subclass of another when it should have had an instance variable of that class as a component. For example, some object-oriented user-interface systems make windows be a subclass of Rectangle, since they are rectangular in shape. However, it makes more sense to make the rectangle be an instance variable of the window. Windows are not necessarily rectangular, rectangles are better thought_of as geometric values whose state cannot be changed, and operations like moving make more sense on a window than on a rectangle.

Behavior can be easier to reuse as a component than by inheriting it. There are at least two good examples of this in Smalltalk-80. The first is that a parser inherits the behavior of the lexical analyzer instead of having it as a component. This caused problems when we wanted to place a filter between the lexical analyzer and the parser without changing the standard compiler. The second example is that scrolling is an inherited characteristic, so it is difficult to convert a class with vertical scrolling into one with no scrolling or with both horizontal and vertical scrolling. While multiple inheritance might solve this problem, it has problems of its own. Moreover, this problem is easy to solve by making scrollbars be components of objects that need to be scrolled.

Most object-oriented applications have many kinds of hierarchies. In addition to class inheritance hierarchies, they usually have *instance hierarchies* made up of regular objects. For example, a user-interface in Smalltalk consists of a tree of views, with each subview being a child of its superview. Each component is an instance of a subclass of View, but the root of the tree of views is an instance of StandardSystemView. As another example, the Smalltalk compiler produces parse trees that are hierarchies of parse nodes. Although each node is an instance of a subclass of ParseNode, the root of the parse tree is an instance of MethodNode, which is a particular subclass. Thus, while View and ParseNode are the abstract classes at the top of the class hierarchy, the objects at the top of the instance hierarchy are instances of StandardSystemView and MethodNode.

This distinction seems to confuse many new Smalltalk programmers. There is

often a phase when a student tries to make the class of the node at the top of the instance hierarchy be at the top of the class hierarchy. Once the disease is diagnosed, it can be easily cured by explaining the differences between the instance and class hierarchies.

Polymorphism **[RGPV /DEC-2013(10),DEC-2010(10)]**

Poly means "many" and *morph* means "form." In the context of object-oriented systems, it means objects that can take on or assume many different forms. *Polymorphism* means that the same operation may behave differently on different classes. Booch defines *polymorphism* as the relationship of objects of many different classes by some common super class; thus, any of the objects designated by this name is able to respond to some common set of operations in a different way. For example, consider how driving an automobile with a manual transmission is different from driving a car with an automatic transmission. The manual transmission requires you to operate the clutch and the shift, so in addition to all other mechanical controls, you also need information on when to shift gears. Therefore, although driving is a behavior we perform with all cars (and all motor vehicles), the specific behavior can be different, and depending on the kind of car we are driving. A car with an automatic transmission might implement its *drive* method to use information such as current speed, engine RPM, and current gear.

Polymorphism allows us to write generic, reusable code more easily, because we can specify general instructions and delegate the implementation details to the objects involved. Since no assumption is made about the class of an object that receives a message, fewer dependencies are needed in the code and, therefore, maintenance is easier. For example, in a payroll system, manager, office worker, and production worker objects all will respond to the compute payroll message, but the actual operations performed by are object specific.

Operations are performed on objects by ``sending them a message'' (The object-oriented programming community does not have a standardized vocabulary. While ``sending a message'' is the most common term, and is used in the Smalltalk and Lisp communities, C++ programmers refer to this as ``calling a virtual function''.) Messages in a language like Smalltalk should not be confused with those in distributed operating systems. Smalltalk messages are just late-bound procedure calls. A message send is implemented by finding the correct method (procedure) in the class of the receiver (the object to which the message is sent), and invoking that method. Thus, the expression *a + b* will invoke different methods depending upon the class of the object in variable *a*.

Message sending causes polymorphism. For example, a method that sums the elements in an array will work correctly whenever all the elements of the array understand the addition message, no matter what classes they are in. In fact, if array elements are accessed by sending messages to the array, the procedure will work whenever it is given an argument that understands the array accessing messages.

Polymorphism is more powerful than the use of generic procedures and packages n Ada. A generic can be instantiated by macro substitution, and the resulting procedure or package is not at all polymorphic. On the other hand, a Smalltalk object can access an array in which each element is of a different class. As long as all the elements understand the same set of messages, the object can interact with the elements of the array without regard to their class. This is particularly useful in windowing systems, where the array

could hold a list of windows to be displayed. This could be simulated in Ada using variant records and explicitly checking the tag of each window before displaying it, thus ensuring that the correct display procedure was called. However, this kind of programming is dangerous, because it is easy to forget a case. It leads to software that is hard to reuse, since minor modifications are likely to add more cases. Since the tag checks will be widely distributed through the program, adding a case will require wide-spread modifications before the program can be reused

S.NO	RGPV QUESTION	YEAR	MARKS
1.	How is polymorphism achieved at compile time and run time	Dec,2013	10
2.	What is polymorphism and what are various types of it?	Dec,2010	10

UNIT 1/LECTURE 9

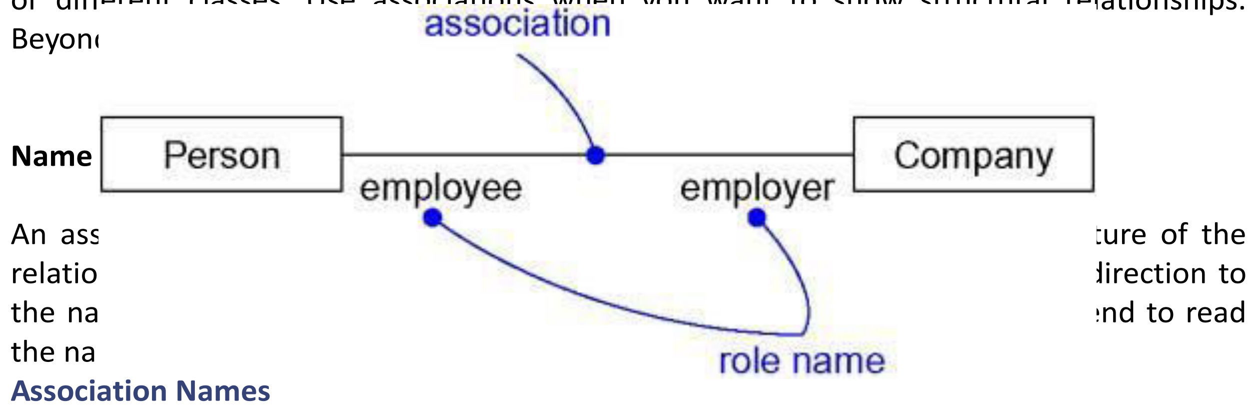
link and association **[RGPV /DEC-2009(10),DEC-2008(8)]**

Association

Associations and dependencies (but not generalization relationships) may be reflective

An *association* is a structural relationship that specifies that objects of one thing are connected to objects of another. Given an association connecting two classes, you can navigate from an object of one class to an object of the other class, and vice versa. It's quite legal to have both ends of an association circle back to the same class. This means that, given an object of the class, you can link to other objects of the same class. An association

that connects exactly two classes is called a binary association. Although it's not as common, you can have associations that connect more than two classes; these are called n-ary associations. Graphically, an association is rendered as a solid line connecting the same or different classes. Use associations when you want to show structural relationships. Beyon



Name

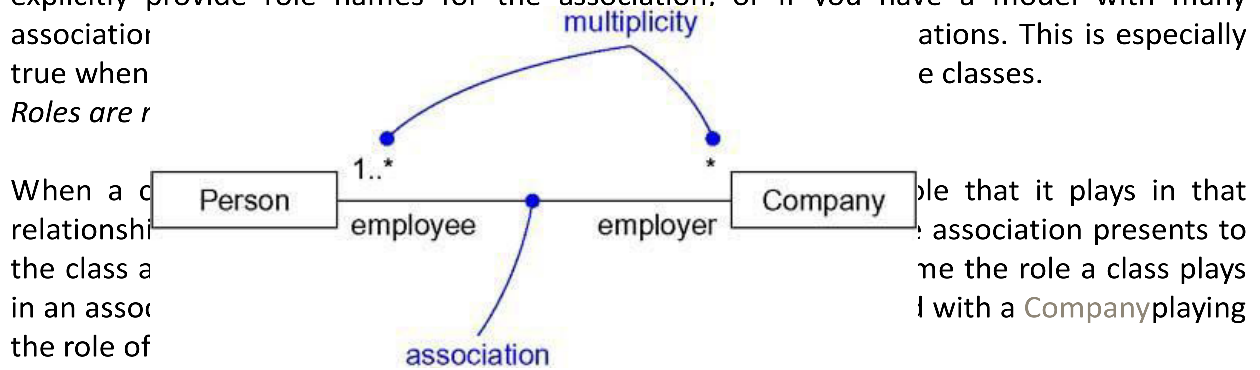
An ass ture of the relatio direction to the na end to read the na

Association Names

Note

Although an association may have a name, you typically don't need to include one if you explicitly provide role names for the association, or if you have a model with many associatio ations. This is especially true when e classes.

Roles are r



When a c ble that it plays in that relationshi association presents to the class a ne the role a class plays in an asso with a Company playing the role of

Roles

Note

The same class can play the same or different roles in other associations.
An instance of an association is called a link

Multiplicity

An association represents a structural relationship among objects. In many modeling situations, it's important for you to state how many objects may be connected across an instance of an association. This "how many" is called the multiplicity of an association's role, and is written as an expression that evaluates to a range of values or an explicit value. When you state a multiplicity at one end of an association, you are specifying that, for each object of the class at the opposite end, there must be that many objects at the near end. You can show a multiplicity of exactly one (1), zero or one (0..1), many (0..*), or one or more (1..*). You can even state an exact number (for example, 3).

Multiplicity

Note

You can specify more complex multiplicities by using a list, such as 0..1, 3..4, 6..*, which would mean "any number of objects other than 2or 5."

ASSOCIATIONS

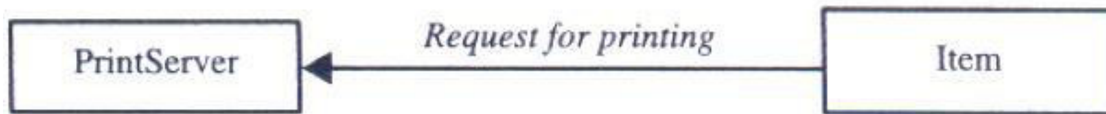
Association represents the relationships between objects and classes. For example, in the statement "a pilot *can fly* planes" (Figure 2.7) the italicized term is an association. Associations are bidirectional; that means they can be traversed in both directions, perhaps with different connotations. The direction implied by the name is the forward direction; the opposite direction is the inverse direction. For example, *can fly* connects a

many
dinality
constrains the number of related objects and often is described as being "one" or 'many," Generally, the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of cylinders in an engine is four, six, or eight. Consider a client-account relationship where one client can have one or more accounts and vice versa (in case of joint accounts); here the cardinality of the client-account association is many to many.

Consumer-Producer Association

•

A special form of association is a consumer-producer relationship, also known as a *client-server association* or a *use relationship.* The *consumer-producer relationship* can be viewed as one-way interaction: One object requests the service of another object. The object that makes the request is the consumer or client, and the object that receives the request and provides the service is the producer or server.



Association represents the relationship among objects, which is bidirectional.

The consumer/producer association.
For example, we have a print object that prints the consumer object. The print producer provides the ability to print other objects. Figure 2.8 depicts the consumer-producer association

Need for object oriented approach

Object Oriented Methodology closely represents the problem domain. Because of this, it is easier to produce and understand designs.
The objects in the system are immune to requirement changes. Therefore, allows changes more easily.
Object Oriented Methodology designs encourage more re-use. New applications can use the existing modules, thereby reduces the development cost and cycle time.
Object Oriented Methodology approach is more natural. It provides nice structures for thinking and abstracting and leads to modular design

S.NO	RGPV QUESTION	YEAR	MARKS
1	Explain links and association with suitable example	Dec,2009	10
2	Explain links and association with example, also give the importance of association	Dec,2008	8