

# CS 201: Object Oriented Design and Programming

Pramit Mazumdar

# Challenges in Software Engineering

- Why is it important to study object orientation?
- How does it correlates with solving real life problems?
- How it can be used in the design of a complex software?

# Challenges in Software Engineering

- Complex Software:
  - Analysis
  - Design
  - Implementation
  - Maintenance
- Object oriented programming languages are of great help
- Such as Java, C++, Python etc.
- Advantages: OOPLs have definition, creation, destruction, management, etc.

# Challenges in Software Engineering

- However, mere use of a OOPL does not guarantee the above
- There are various level of problems that are faced during software engineering
  - Problems encountered by the software systems
  - Problems encountered by the developers
  - Problems encountered by the users of the software systems
- Solution to such problems cannot be addressed by just learning and using the OOPL !!!

# Challenges in Software Engineering

- Most of the problems start at the specification level
- Where users fail to understand what they want
- Understanding the expectations and clearly describing them to the developer (enterprise) is important
- Sometimes softwares are specific to a person or organization
- In this case the specifications are defined by users

# Challenges in Software Engineering

- Similarly there are software which are generic in nature
- For example MS PowerPoint/ Excel/ Word, etc.
- This is more challenging as multiple users round the globe have multiple requirements and they are fused together meticulously to give a proper user experience
- In both the scenarios; specification, design, analysis, implementation, maintenance, etc. is important.

# Challenges in Software Engineering

- Typically a software engineer is responsible for engineering a software that can be used and liked by the end users
- Remember: Software is DEVELOPED not CONSTRUCTED
- This is unlike any other Engineering domain !!
- Civil = constructs buildings
- Mechanical = constructs automobiles
- Electrical = constructs power plants
- Software = *develops a software !!*

# Challenges in Software Engineering

- According to Standish Group's Annual CHAOS report, 66% of technology projects (based on the analysis of 50,000 projects globally) end in partial or total failure\*
- Just one in three software projects are considered truly successful with large projects most at risk of failure.

	SUCCESSFUL	CHALLENGED	FAILED 
Large	8%	26%	41%
Medium	9%	26%	31%
Moderate	21%	32%	17%
Small	62%	16%	11%

# Why Software Fails !!

- COMPLEXITY .... complexity in various domain
- RESOURCES ... creator of many problems
- Changes from Requirements
  - Customers learn from software
  - Requirement changes as Prototype is shown
  - Customers do not know what they want !!!
  - Business environment changes
  - Business condition changes
  - Process of Business changes
- Changes from Technology
  - Software is not standalone
  - It depends upon hardware, OS, network component, etc.
  - Over time technology evolves
  - The system from which one started the project may end up into an obsolete system!!

# Why Software Fails !!

## ■ Changes from People

- Usually the process of constructing something can be automated
- However in software the process of development is difficult to automate
- For example it is difficult to automate writing of codes
- Ultimately it is the people or developers who write the code
- Complexity starts from here
  - Developer changes
  - Multiple developers may not agree to an approach
  - One may not follow the approach followed by an ex-developer of the same project
  - Etc.

# Software Engineering

- Is Software engineering
  - A manufacturing process, or
  - A designing of a manufacturing or service oriented process
- The process of designing is an innovative process
- It is not a predictive or automated process
- Requires a great deal of Communication / Understandability / Innovation, etc.

# Target

- Software Development -> Software Construction
- In future can we
  - Construct a software
  - Confidence that the constructed software would work
  - The process of construction would be predictable
  - The quality of the software would be predictable
  - The cost of software development would be predictable

# Target

- Current SOA says that to achieve the above we need
  - Structured analysis
  - Modelling
  - Design
  - Implementation
  - Adherence to good practices.

# Why this course

- Object Oriented Analysis and Design
- It is a **precursor** to the previously mentioned requirements
- Understanding the basics of OOAD can help in formulating the development process of a software
- There are topics in OOAD which can directly help to address problems faced by developers while developing a software.

# Types of Software

- Personal or limited use Software
- Industrial-strength Software
- Personal or Limited use Software
  - Limited set of behaviors
  - Not very complex
  - Developed, maintained and used by the same person or group
  - Short life span
  - Demonstration purpose
  - Not expected to be lasting forever
  - Presence of small bugs is not of concern
  - Lack of quality.

# Types of Software

- Industrial-strength Software
- It refers to a software which have rich set of behaviors
- Well planned in accordance with the user requirements
- Works with limited resources
- Maintains the integrity of millions of records while allowing concurrent updates and queries
- Commands and controls a real-world problem
- Long life-span
- Complexity of such software exceeds the human intellectual capacity.

# Software Complexities

- The complexity of the problem domain
- The difficulty of managing the development process
- The flexibility possible through a software
- The problems of characterizing the behavior of discrete systems.

# Problem domain

- Domains are difficult to understand
- Functional requirements are complex to understand
- Non-functional requirements (non-evident) are also complex to predict in advance
  - Usability
  - Performance
  - Cost
  - Survivability
  - Reliability
  - Users do not understand the non-functional requirements
  - Implicit in nature !!

# Problem domain

- Communication gap between Users and Developers
  - The greatest bottleneck in a software development process
  - Users cannot express and Developers cannot understand
  - Different perspective of the problem in hand (different assumptions).
- Changing or evolving requirements
  - More requirements comes up with better understanding of the prototype
  - Developers often gets enlightened through the process of development.

# Managing the Development Process

- Illusion of Simplicity
  - Hide external complexity
  - Code bases get large
  - Software should give the illusion to the user that the task which he/she performed is extremely easy
  - Whereas the backend code base is having a very complex logic, which is abstracted from the user !
- Increase in code-base requires involvement of more people in the project
- As it is impossible for a single person to run the large code-base.. That's why a Software development process requires a Team !!

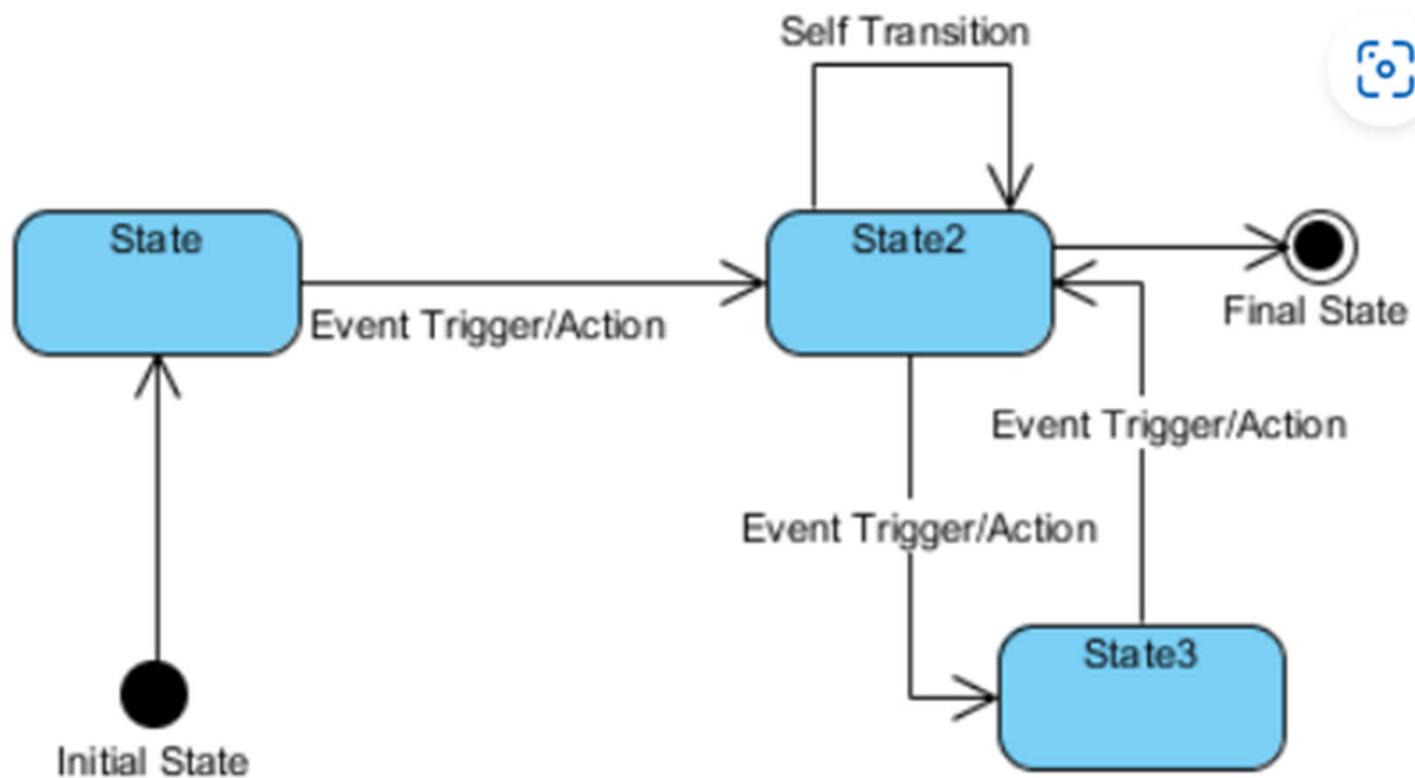
# Flexibility possible through a Software

- Components required to build a Software is not readily available
- There is no standardization on how to build a software
- Often the components required are also software
- Interestingly, software is such an element which may require additional features after it has been deployed
- Such flexibility is Excellent, but brings a lot of complexity.

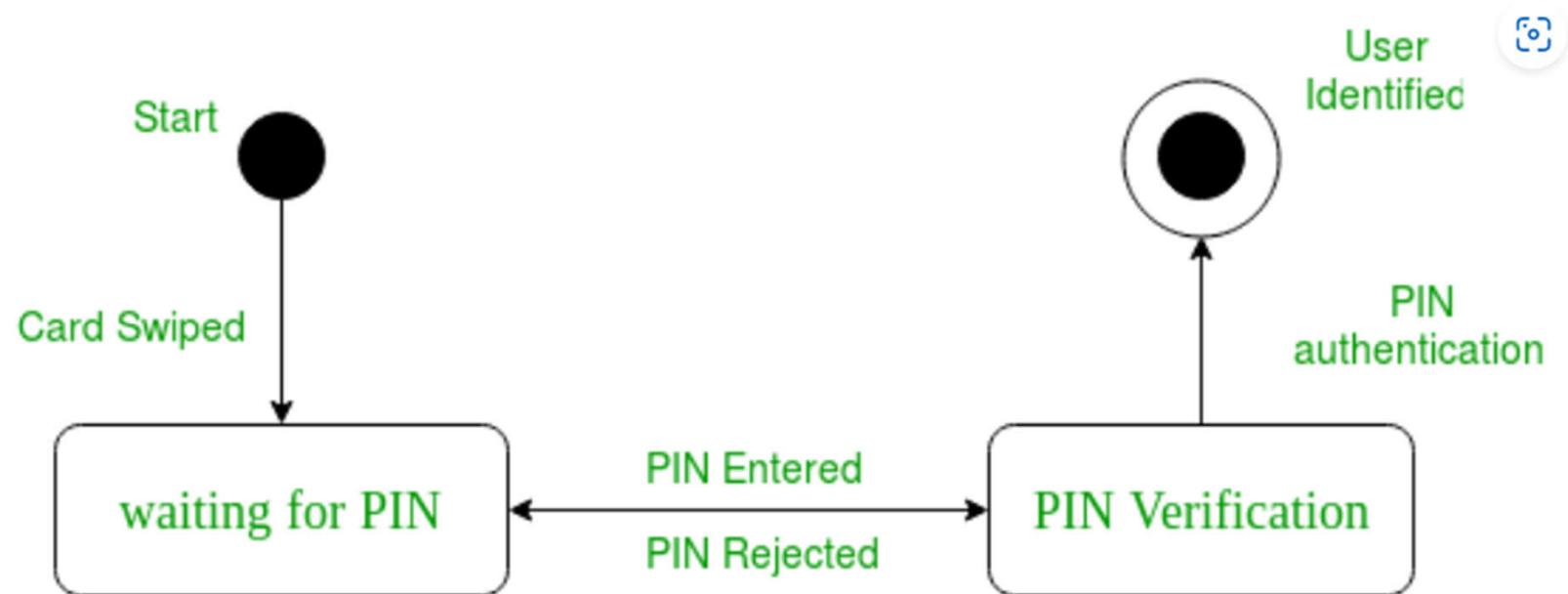
# Behavior of Discrete system

- Discrete system changes its state abruptly at discrete points, consists of multiple states
- Software are built and executed on digital computers – thus on Discrete systems
- A software can have infinite number of states
- Various activities to be performed by the software corresponds to the different states
- These states are often interactable and influenced by external factors.

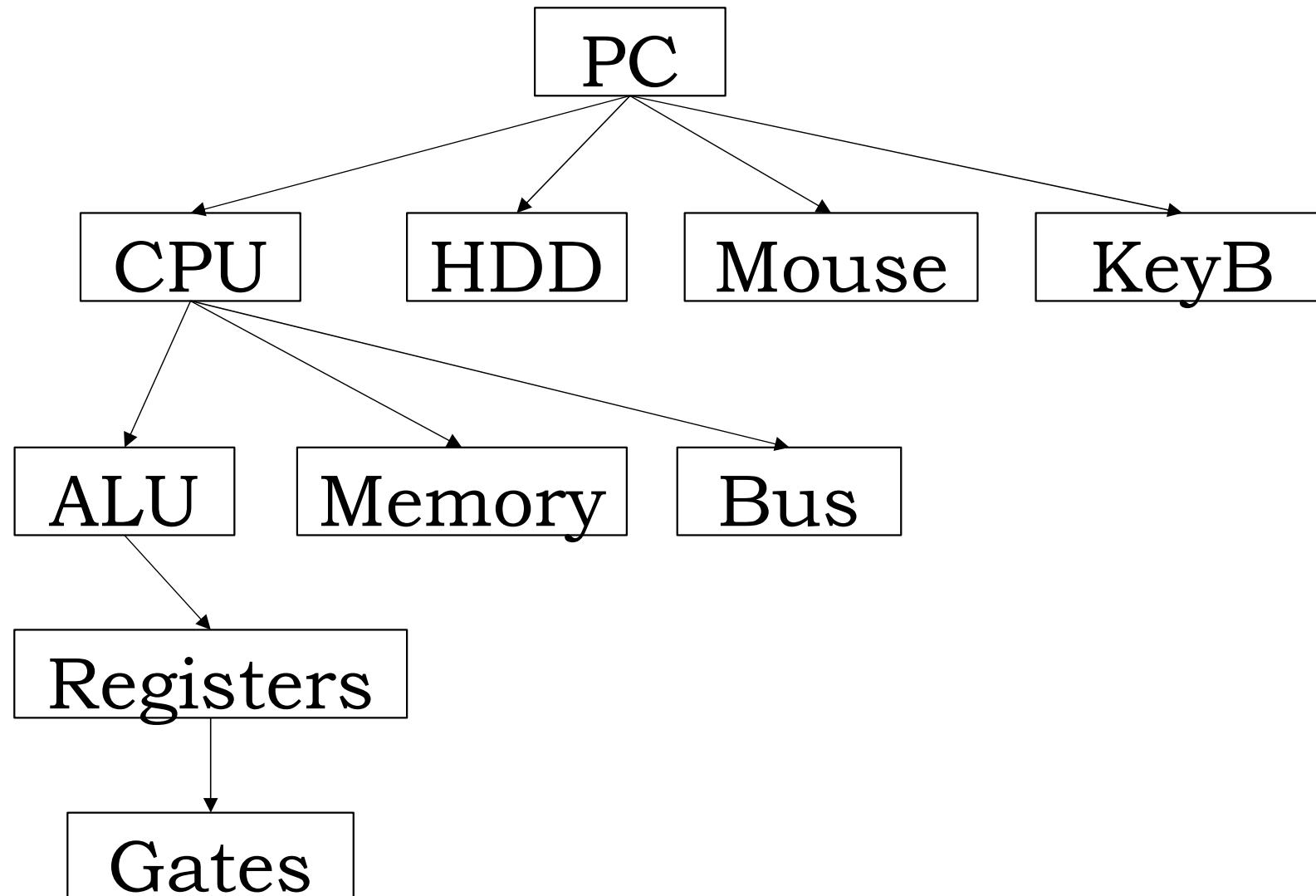
# Behavior of Discrete system



# Behavior of Discrete system



# Complex systems are Hierarchic



# Complex systems are Hierarchic

- Each level of hierarchy represents a Layer of abstraction
- Task of CPU is independently understood
- How CPU performs a task is not known
- ALU -> Registers -> Gates have their own tasks defined
- But they do not know the overall idea or the bigger picture
- This is known as **Emergent Behaviour.**

# Complex systems are Hierarchic

- Layers in the hierarchy shows emergent behaviour
- Behaviour of the whole is greater than the sum of its parts
- Behaviour of CPU is greater than the sum of its parts (ALU, Memory, etc.)
- Interlinked behaviour of the individual parts in a lower level of hierarchy defines the broad task of a component (at higher level of hierarchy).

# Complex systems are Hierarchic

- All systems are composed of interrelated sub-systems
- Sub-systems are composed of sub-sub systems ..
- Lowest level sub-systems are composed of elementary components
- All systems are parts of larger systems
- Users only understands a system which is at a higher level of hierarchy !!

# Five attributes of a Complex system

- Hierarchic structure
- Relative primitives
  - Primitives – section of code which can be used to develop more sophisticated modules
- Strong separation of concerns among the elements
- Common patterns across the elements
- Stable intermediate forms.

# Relative properties

- Primitives are chosen by the developer
- It is a Subjective choice
- Strongly depends on the experience and expertise of the designer
- A primitive chosen by a developer may be completely abstract for another.

# Separation of Concerns

- Hierarchic systems are
  - Decomposable = consists of independent elements, so can be decomposed
  - Nearly decomposable = consists of all non-independent elements, so cannot be decomposed
- Linkages /associations / dependencies between elements in a hierarchy is important
- Intra-component linkage involves internal structure of components. Interaction is stronger within component
- Inter-component linkage involves interaction among components. Interaction is weaker across component.

# Common Patterns

- Processor is found in computers, cars, mobiles, etc.
- If we understand the working of a processor, it is easier to map how computers, cars, mobiles, etc. performs
- Most complex systems have common patterns
- Such common patterns are one important aspect of OOAD
- Common patterns can become a primitive.

# Stable Intermediate Forms

- It is extremely difficult to design a complex system correctly in one go
- Iterative refinement is the approach considered to be helpful in a software development process
- A complex system is always build from a simple system which worked .. Prototypes !!.

# Types of Complexities

- Organized (Canonical Form)

- Decomposition hierarchy
  - Abstraction hierarchy
  - Class and object structure
  - Canonical form

- Disorganized

- Limited human capacity

# Organized Complexity

- We are able to find a structure
- We are able to find some common patterns
- Easy to device tools and mechanism to approach the problem in a better way
- This leads to a standardized form or a canonical form of a complex system

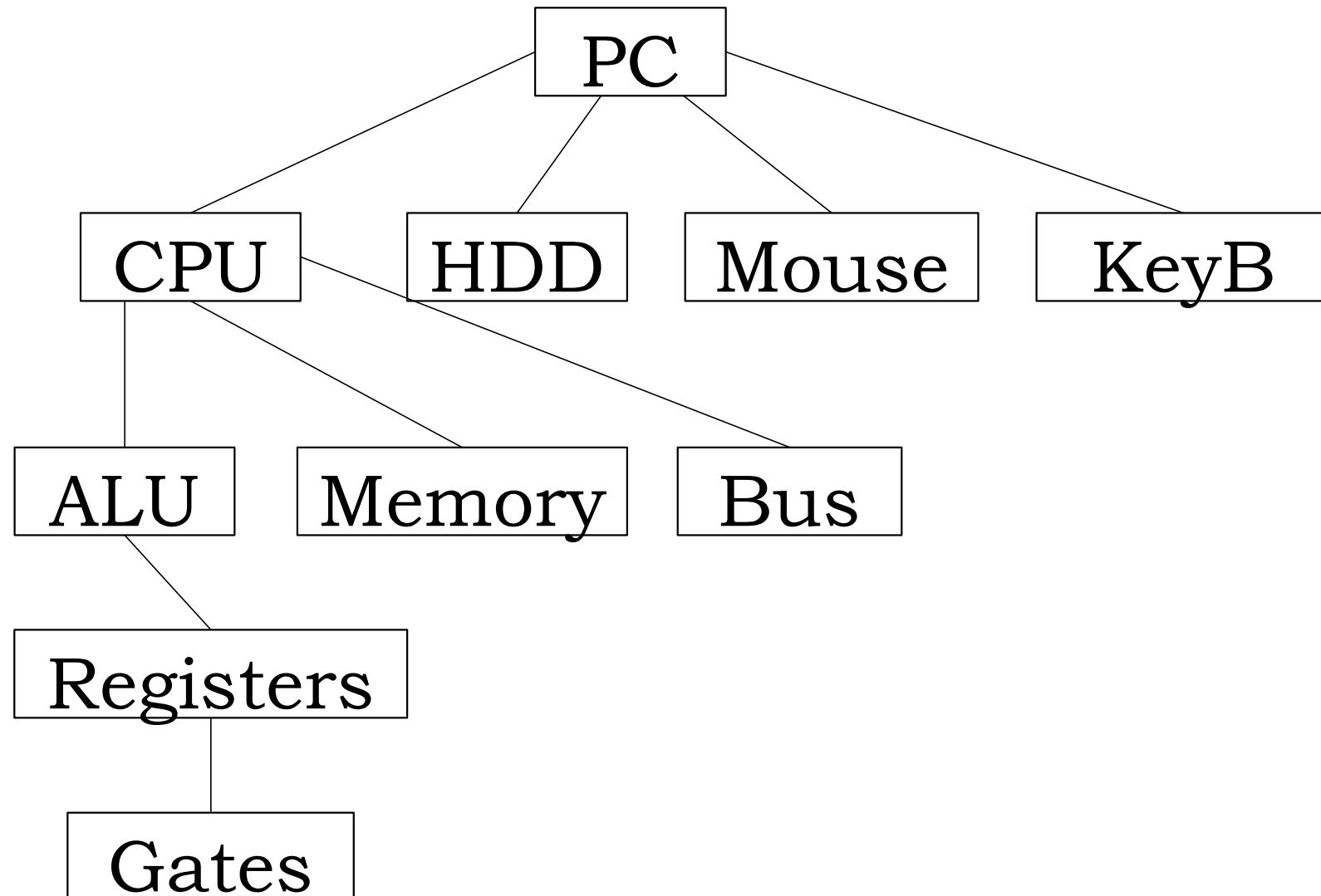
# Limitation of Human Capacity for Dealing Complexity

- Huge chunk of information is required to be processed for designing a complex system
- Human processing capacity is limited
- Processing speed is limited
- Group of individuals as a team may help
- Groups brings new challenges with them
- This leads to Disorganized Complexity.

# Types of Hierarchy

- We know that complex systems are hierarchical in nature
- Hierarchies that are mostly used in a OOAD are;
  - **Decomposition = part-of / has-a**
  - **Abstraction = is-a**
- Decomposition refers to **part-of / has-as** relationship between components
- Abstraction refers to **is-a** relationship between components.

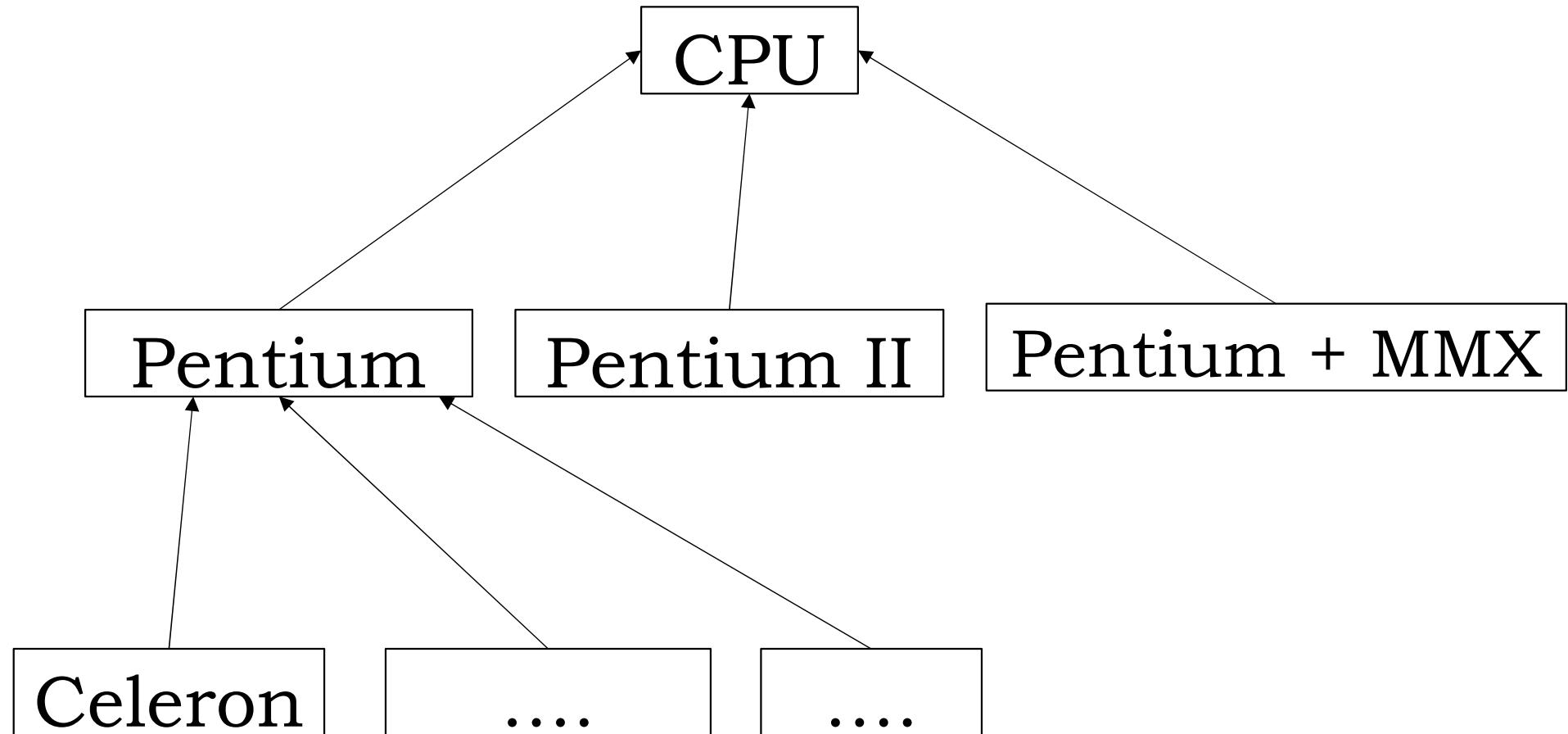
# Decomposition Hierarchy



# Decomposition Hierarchy

- PC can be broken down to components such as CPU, HDD, Mouse, Keyboard, etc.
- CPU can also be broken down to ALU, Memory, Bus, etc.
- If such is the relationship between the components in a hierarchy then we refer it as Decomposition hierarchy
- PC **has-a** CPU. PC **has-a** HDD. CPU **has-a** ALU.

# Abstraction Hierarchy



# Abstraction Hierarchy

- Specialization of CPU is not mentioned in previous example
- We never mentioned the type of CPU or the type of processor
- CPU can be
  - Pentium
  - Pentium + MMX
  - Celeron
- All processors have different details
- But they share the same functionality of a CPU.

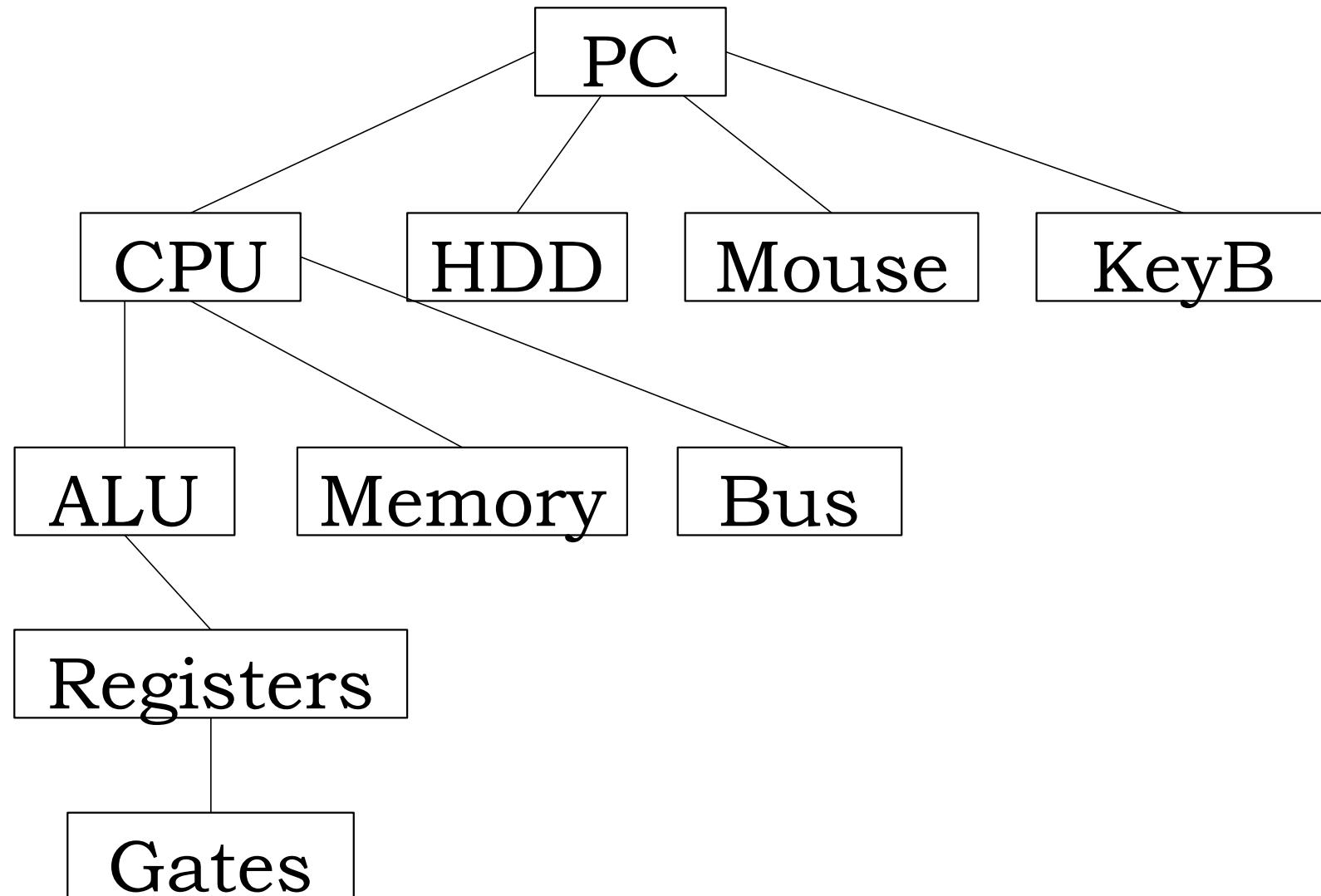
# Abstraction Hierarchy

- Pentium **is-a** CPU
- Celeron **is-a** Pentium + MMX
- Pentium + MMX **is-a** Pentium
- Pentium + Multimedia extension is similar to a Pentium with some added functionality
- So Pentium has many variations but they share the same concept or functionality to some extend
- Such hierarchy among components are known as the Abstraction hierarchy.

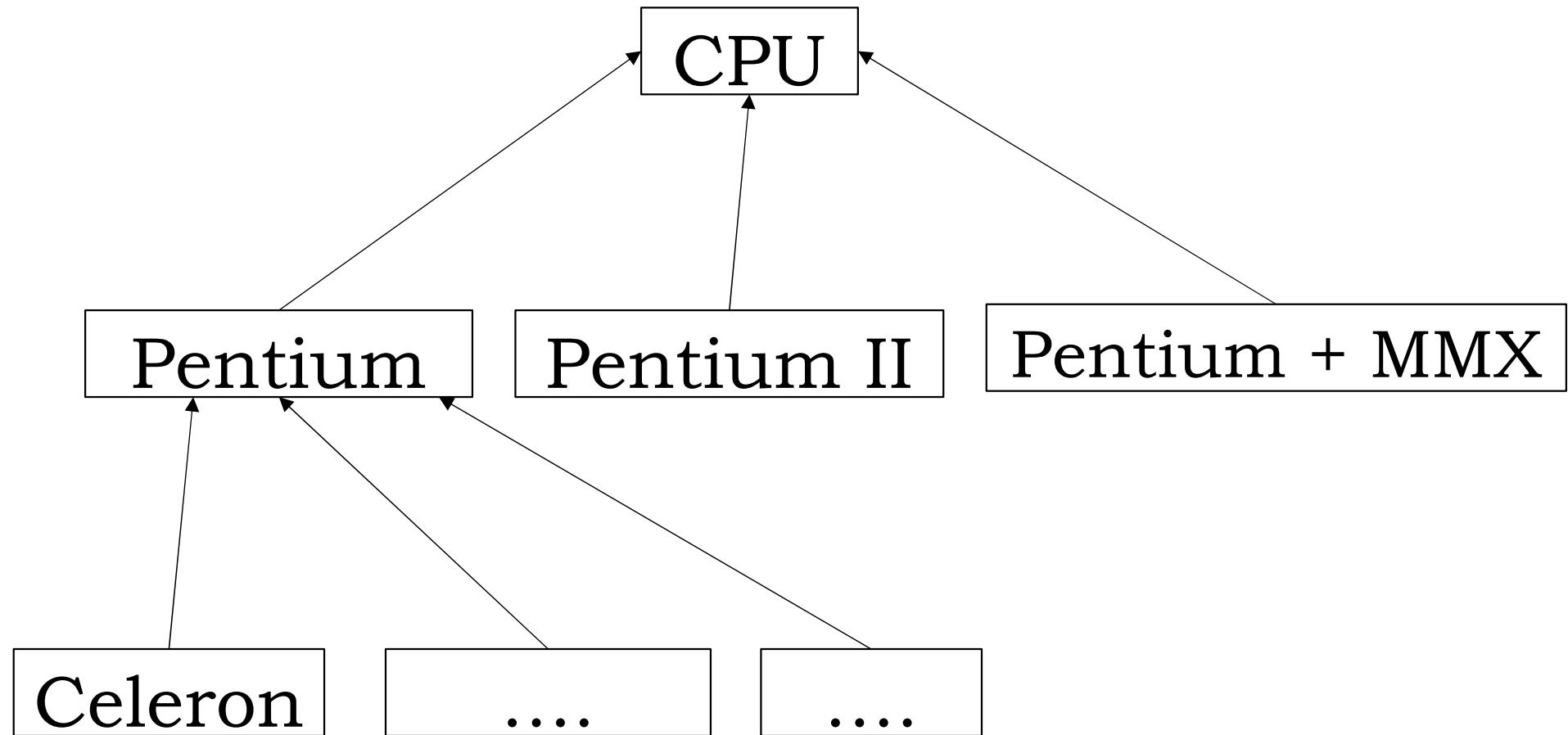
# Class and Object Structures

- Hierarchies are referred to as:
  - Class structure: Abstraction (IS-A)
  - Object structure: Decomposition (HAS-A)
- Complex systems representation has two orthogonal access; one about abstraction hierarchy and the other talks about the decomposition hierarchy

# Decomposition Hierarchy



# Abstraction Hierarchy



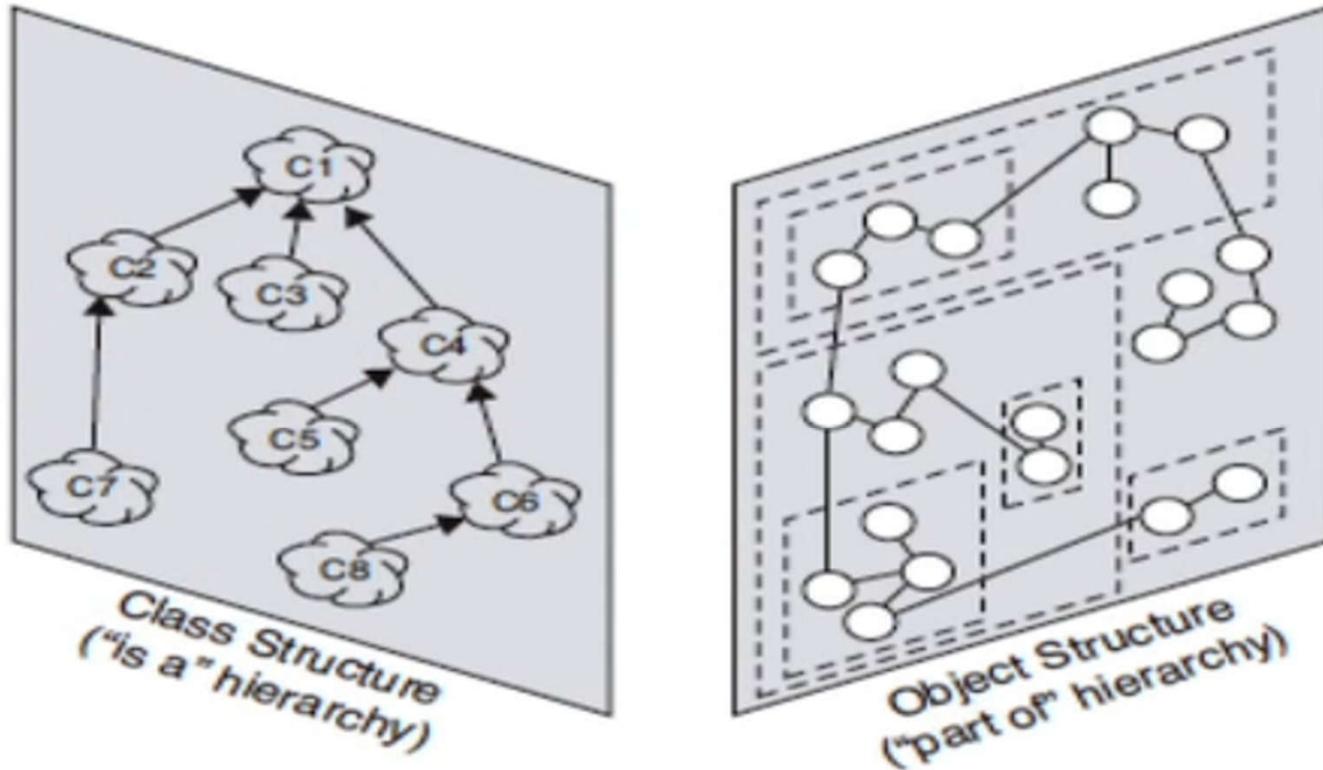
# Abstraction Hierarchy

- Specialization of CPU is not mentioned in previous example
- We never mentioned the type of CPU or the type of processor
- CPU can be
  - Pentium
  - Pentium + MMX
  - Celeron
- All processors have different details
- But they share the same functionality of a CPU.

# Abstraction Hierarchy

- Pentium **is-a** CPU
- Celeron **is-a** Pentium + MMX
- Pentium + Multimedia extension is similar to a Pentium with some added functionality
- So Pentium has many variations but they share the same concept or functionality to some extend
- Such hierarchy among components are known as the Abstraction hierarchy.

# Class and Object Structures



*The Key Hierarchies of Complex Systems*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

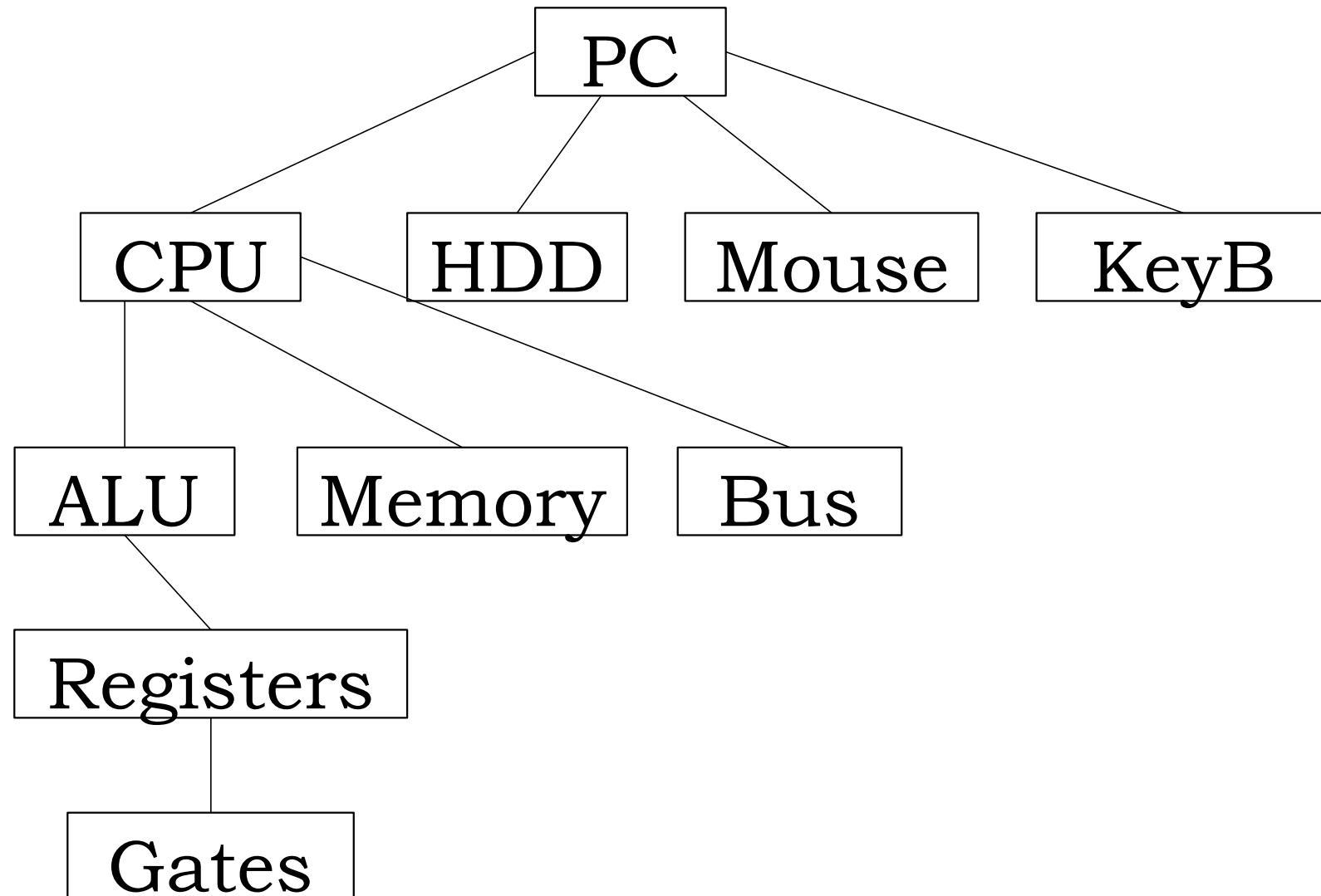
# Class – Abstraction – IS-A relationship

- Class structure shows IS-A relationship
- Class structure shows Abstraction
- Example:
  - C1 = CPU
  - C2 = Pentium
  - C3 = Mac
  - C7 = Celeron
- IS-A relationship is represented by →
- Celeron → Pentium

# Class – Abstraction – IS-A relationship

- Celeron IS-A Pentium
- Celeron includes the properties that Pentium has
- Pentium may have more properties than Celeron
- Pentium can be defined by the various properties that Celeron has
- Similarly, Pentium IS-A CPU means Pentium has all the properties that a CPU has (or the properties which define a CPU)
- Thus when we look into the system in terms of the commonality of behavior then we say that it is a Class structure representing Abstraction.

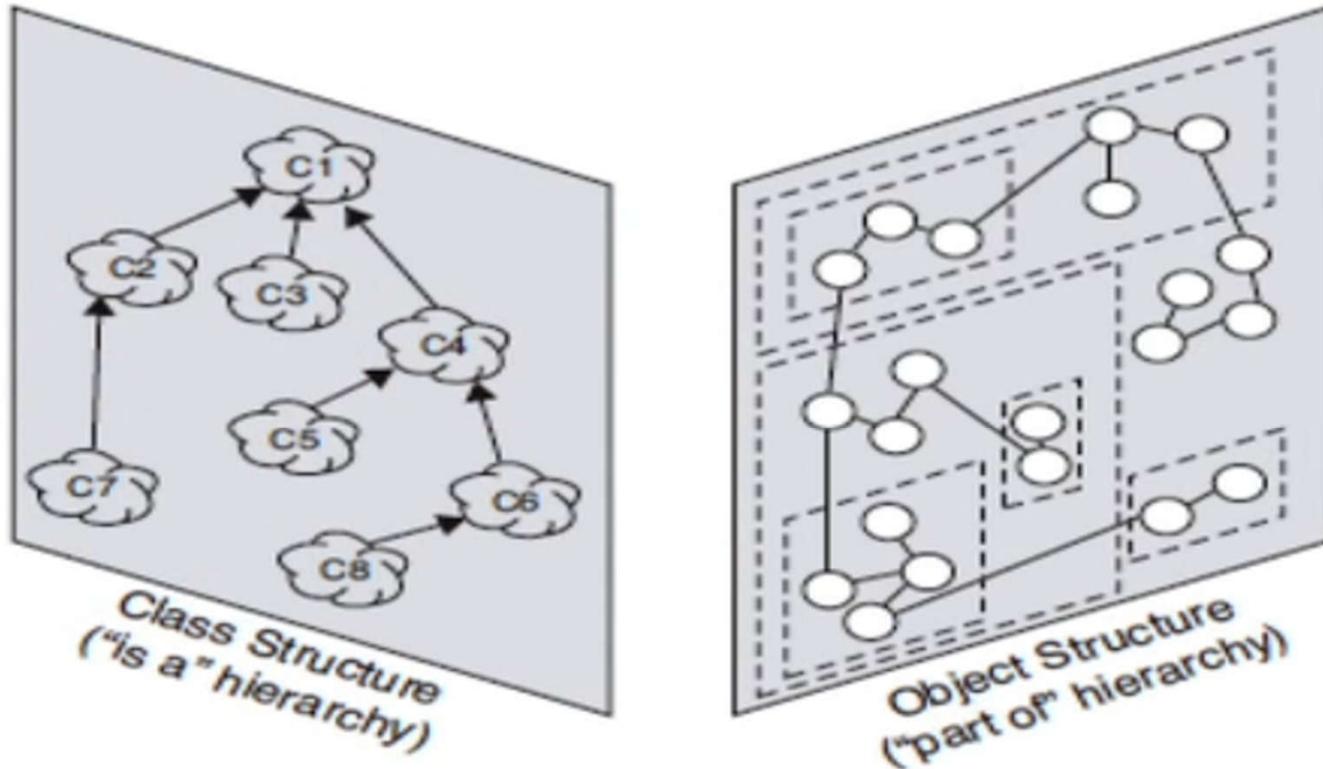
# Decomposition Hierarchy



# Decomposition Hierarchy

- PC can be broken down to components such as CPU, HDD, Mouse, Keyboard, etc.
- CPU can also be broken down to ALU, Memory, Bus, etc.
- If such is the relationship between the components in a hierarchy then we refer it as Decomposition hierarchy
- PC **has-a** CPU. PC **has-a** HDD. CPU **has-a** ALU.

# Object – Decomposition – HAS-A relationship



*The Key Hierarchies of Complex Systems*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

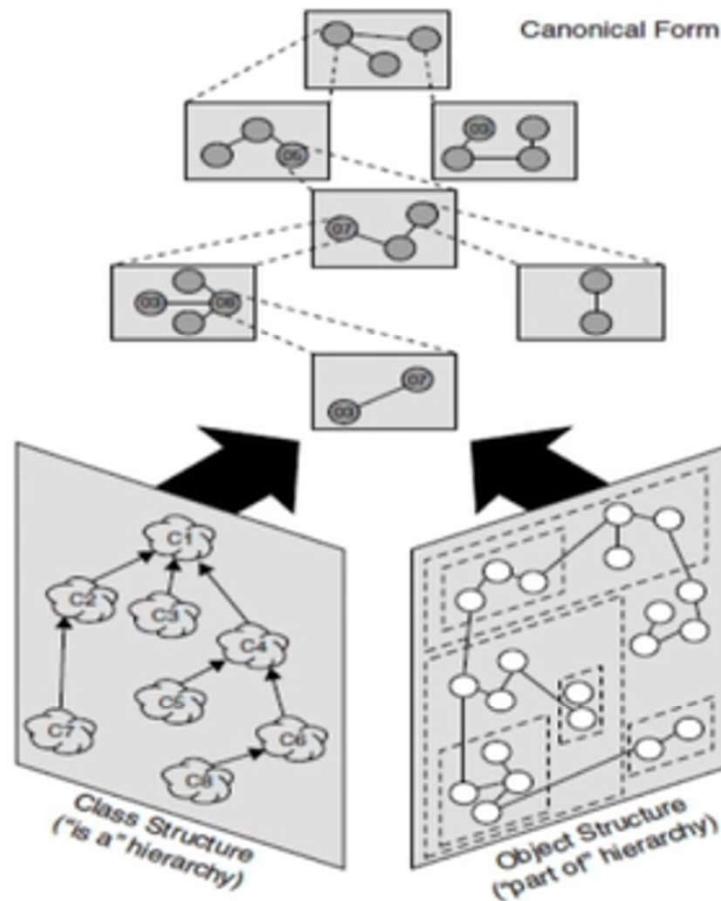
# Object – Decomposition – HAS-A relationship

- Dotted lines represents an entity
- An entity consists of multiple sub-entities
- Thus the bigger entity can be broken down or decomposed into smaller or sub-entity
- So we can look into the system in the form of which is the bigger component and what are the sub-component with which it is developed.

# Canonical Form of a Complex System

- System Architecture shows how a Complex system looks like
- In term of class abstraction and object decomposition
- If a complex system can be represented in terms of the five attributes
  - *Hierarchic structure*
  - *Relative primitives*
  - *Separation of concerns*
  - *Common patterns*
  - *Stable intermediate forms*
- Then we can say that the complex system is being designed in a CANONICAL FORM.

# Canonical Form of a Complex System



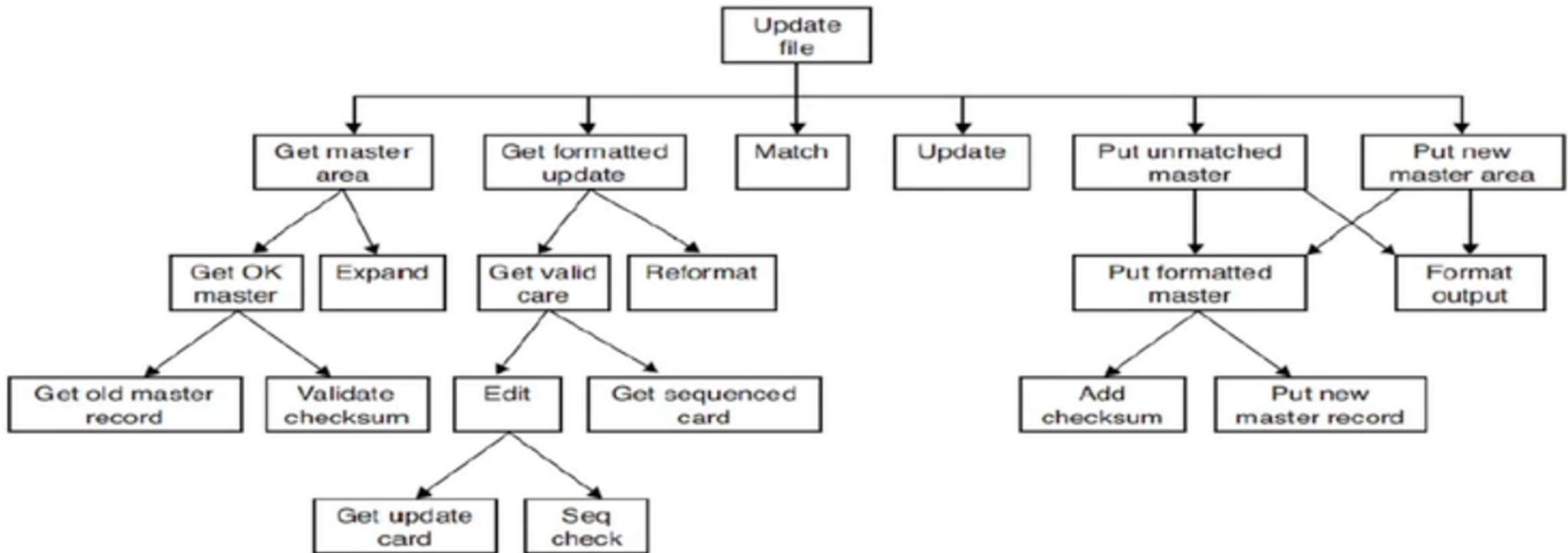
*The Canonical Form of a Complex System*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

# The Role of Decomposition

- We need to understand that given a complex system how should be approach to design a decomposition hierarchy
- There are two approaches:
  - Algorithmic approach for Decomposition
  - Object-oriented approach for Decomposition
- Algorithmic approach
  - Top-down structured design
  - Divide-and-conquer
  - Decompose the problem into key processing steps.

# Algorithmic Decomposition



## Algorithmic Decomposition: Structure Chart

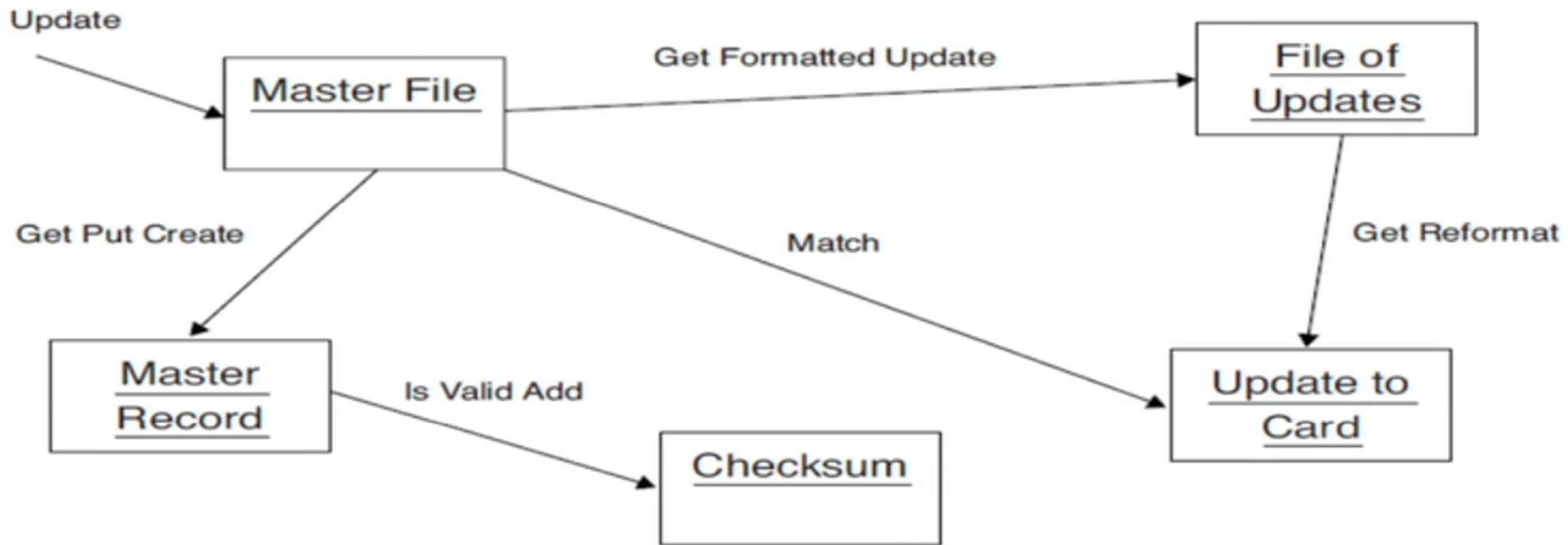
Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

- Involves structured decomposition of individual components.

# Object-oriented Decomposition

- Here we do not look into the task in hand
- We look for the key abstractions that are required in the task
- Decompose the system based on the identified abstractions
- The abstractions can also be called as Agents.

# Object-oriented Decomposition



**Object-Oriented Decomposition**

# Object-oriented Decomposition

- Each one of them is a key concept or key abstraction
- Remember the arrow direction
- Master File can *get formatted update* from the File Updates abstraction
- Object-oriented decomposition is distinct w.r.t Algorithmic approach
- Object-oriented approach views the system as **autonomous agents** that collaborate to perform some higher-level behaviour
- “Autonomous agents” = active entities that can behave on their own.

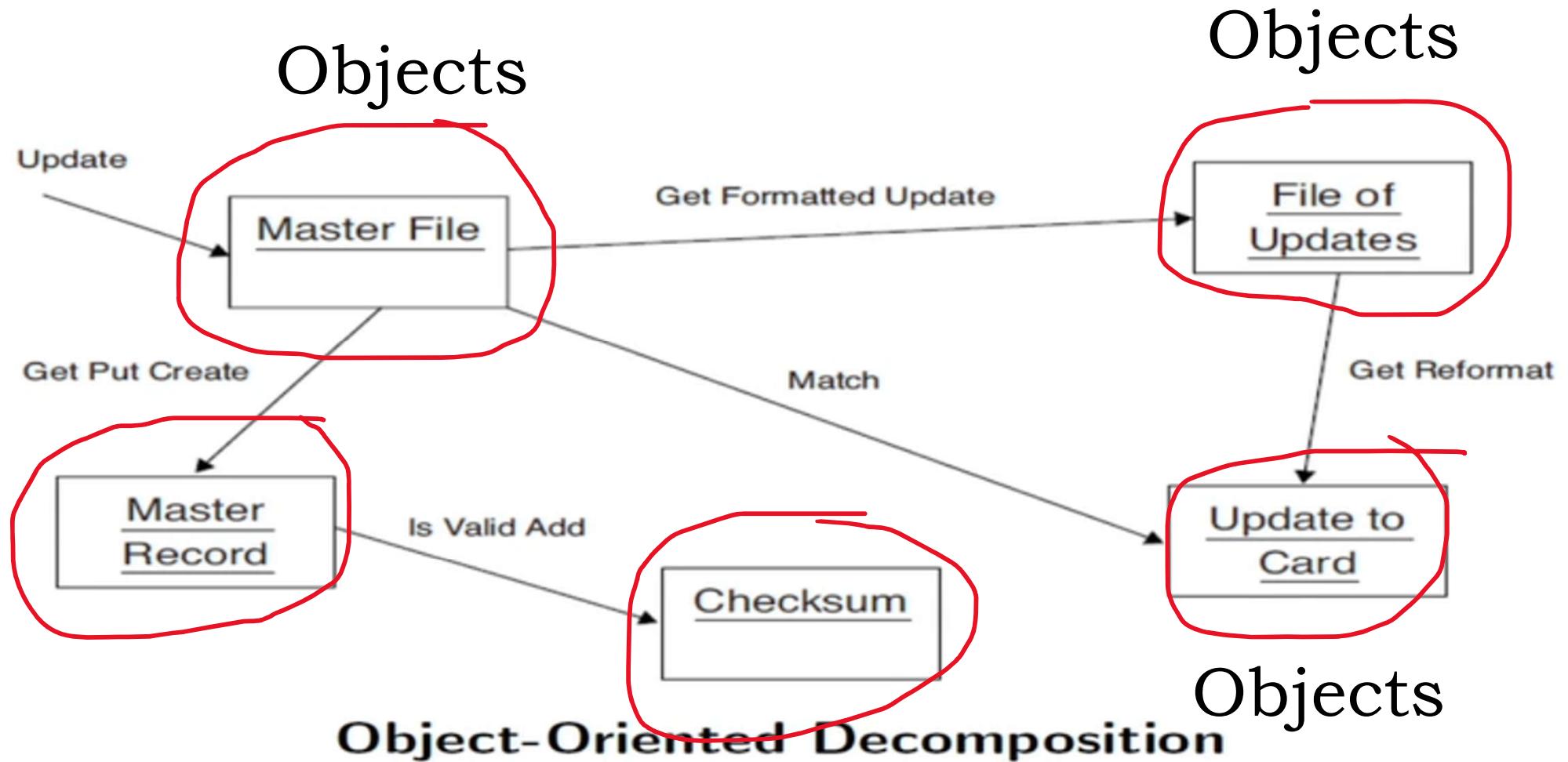
# Object-oriented Decomposition

- Each autonomous entity in the previous example are Objects
- Master File can *get formatted update* from the File Updates object
  - “*get formatted update*” becomes an operation that is associated with the object “*File of Updates*”
  - By calling a function “*get formatted update*” on “*File of Updates*” it creates another object called “*Update to Cards*”
- Each object in an object-oriented decomposition holds its own behaviour
- Objects are tangible entity that exhibits some well-defined behaviour.

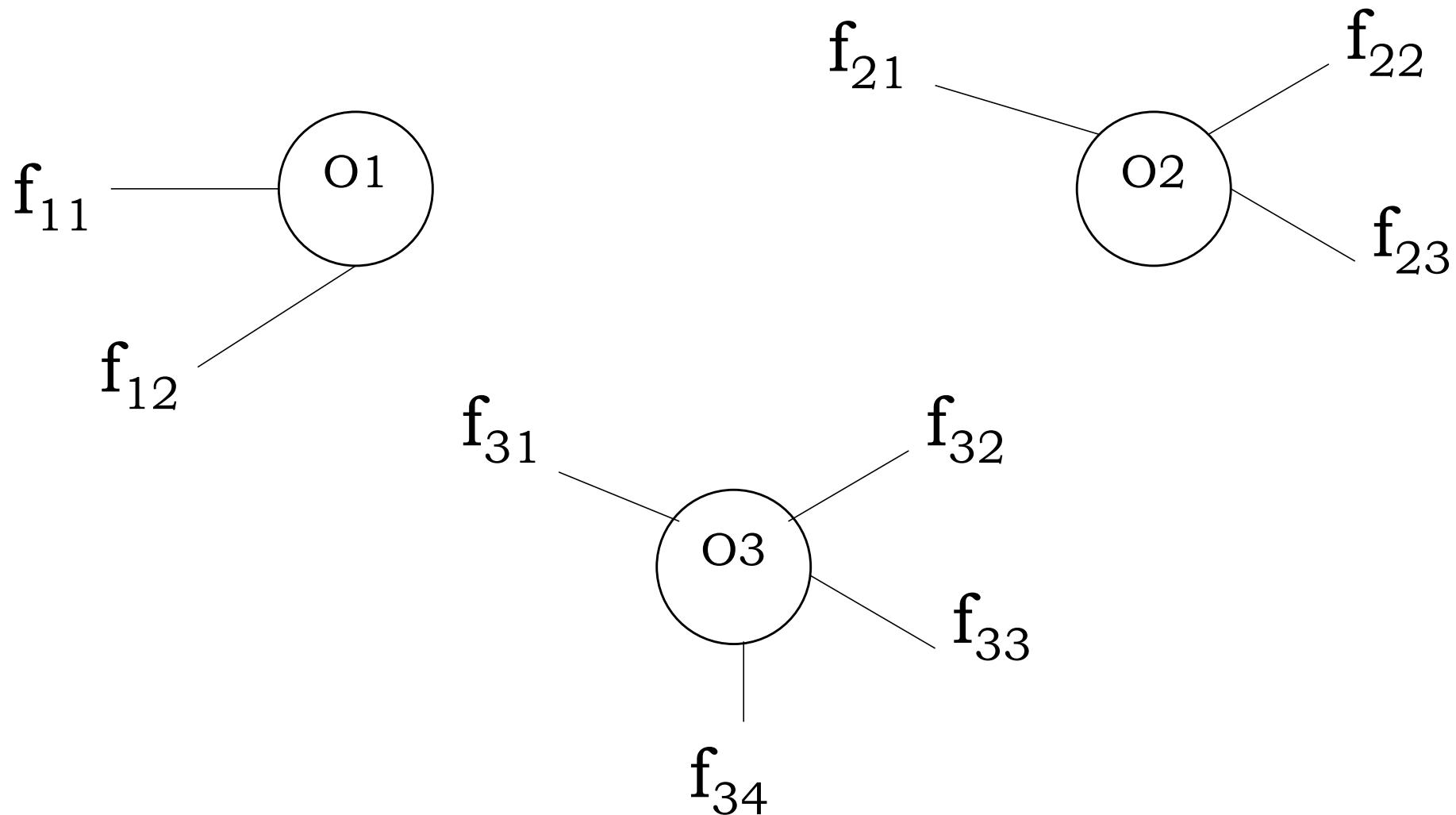
# Object-oriented Decomposition

- Object do things ... or things in a designing process is being done by using Objects .. As they are autonomous and have well-defined behaviour
- Objects communicate among themselves
- They communicate by sending messages among each other
- These are the advantages achieved if the complex system is designed using an object-oriented approach and not an algorithmic approach.

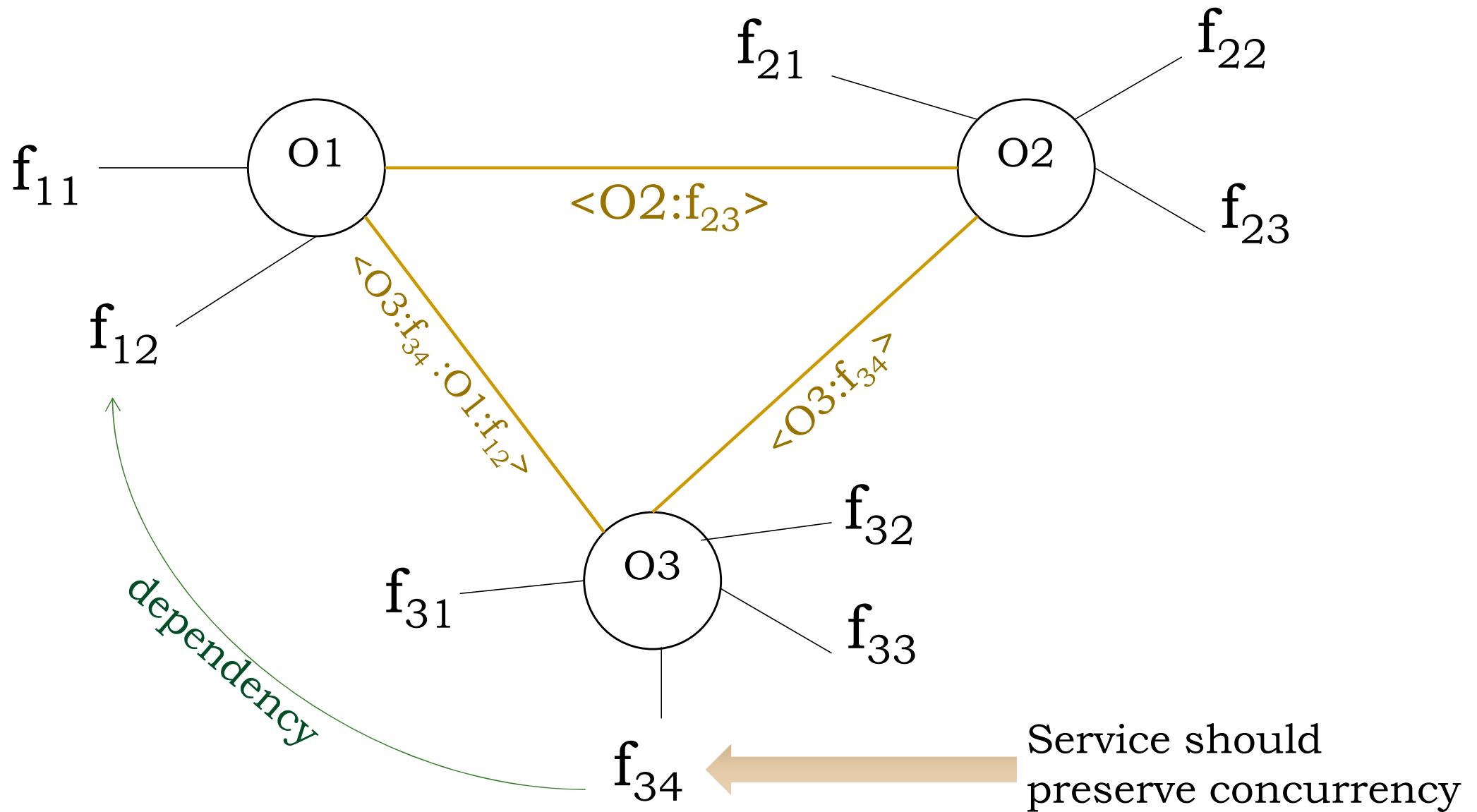
# Object-oriented Decomposition



# Objects provides services



# Objects communicate among themselves



# Object-Oriented Decomposition

- Objects have a certain behaviour
- Objects provides services
- Services are provided by objects may require communication with other objects
- Communication is performed by objects via message sending
- Therefore;
  - Objects can perform multiple tasks or services or operations
  - Objects transmit messages to communicate with other objects.
- Objects are Clients and calling functions/methods are communication between objects.

# Evolution of Design Methods

- Top-down structured design
- Data-driven design
- Object-oriented design

# Top-down structured design

- Oldest and most traditional
- Does not address the issues of data abstraction and information hiding
- Does not provide an adequate means of dealing with concurrency
- Does not scale up well for extremely complex systems
- Is largely inappropriate for use with object-based and object-oriented programming languages.

# Data-driven design

- Mapping system inputs to the outputs
- The above strategy derives the structure of a software system
- It has been successfully applied to multiple complex domains (specially Information management systems)
- However, they are less effective for time-critical events
- In certain Information management systems where time constraint is absolutely important, this design approach cannot be applied
- For example, breaking system in a Car.

# Object-oriented design

- Models software systems as collections of cooperating objects
- Treats individual objects as instances of a class within a hierarchy of classes
- OOAD directly reflects the topology of a high-order programming languages such as Java/ C++.

# Algorithmic vs Object oriented Decomposition

<b>Algorithmic</b>	<b>Object-oriented</b>
Highlights the <b>ordering of events</b>	Emphasizes on <b>Agents</b> (clients / servers)
Typically develops Larger Systems	Yields smaller systems by reusing methodologies
Lower re-usability characteristics	Large re-usability characteristics
Less resilient and more risky to build complex systems	More resilient to change and reduces risk of building complex systems
Unable to reduce complexity	By following the approach of “separation of concerns” is reduces the complexity of a system significantly
Low concurrency	High concurrency and Distributed in nature.

# Abstraction

- Abstraction is another important strategy along with Decomposition which helps in a proper designing of a complex system
- Human brain has limitations: storage problem, unable to process all information, sequentially receiving information, and everything needs to be done in a time-bound way
- Abstraction typically says you look into only those information which you need
- There is no need to know the underlying structure, as it is not immediately needed for processing
- Abstraction helps in chunking the information into utility and underlying architecture.

# Abstraction

- One may look into the utility part of a module and completely ignore the underlying structure of it
- Whereas someone interested in the underlying part do not necessarily need to understand what is the utility or how interaction if performed with the user (for example)
- Thus Abstraction helps in **reducing the semantic load of the information.**

# Recap: Object and Class structure

- Object structure: Illustrates how different objects collaborate with one another through patterns of interaction that we call mechanisms
- Class structure: Highlights common structure and behaviour within a system.
- **Decomposition – Hierarchy – Abstraction**
- These three are the main tool to use when we try to design a complex system
- These three would be frequently used when we learn about other concepts of OOAD.

# What we mean by “Design”?

- We would be using various types of Design approach in OOAD and they would be applied on variety of problems
- Design would always refer to
  - Something that satisfies a given functional specification
  - Helps in addressing the limitation of the target system
  - Meets the requirements on performance and resource of the target system
  - Satisfies restrictions on the design process such as constraints in duration, cost, tools, etc.

# Evolution of Programming Languages

- First-generation (1954-1958)
  - One global data and subprograms
  - Error in one part effects rest of the system
- Second-generation (1959-1961)
  - One global data and subprograms. Subprograms have individual modules
  - Fails in programming-in-large scenario and design
- Third-generation (1962-1970)
  - Structured, Modular, Abstraction
  - Modules were separately compiled and hence no track of errors
- Fourth-generation (1970-1980)
  - Large data handling with Database management

# Evolution of Programming Languages

- Object-orientation (1980-1990)
  - Object-based models
  - Exception handling
  - Task-specific
- Emergence of Frameworks (1990-today)
  - Programming frameworks
  - Used for OOAD for large-scale systems
  - Portable. Threading.
  - Frameworks:
    - J2SE, J2EE, J2ME
    - .Net

# Foundations of Object Models

- OOAD is an Evolutionary development
- The approach is to start small/elementary and proceed to develop the overall system
- Therefore a simpler to further complex in an iterative manner is the process of Evolution in this context
- Following have contributed in this evolution:
  - Computer architecture
  - Programming language
  - Programming methodology
  - Database models
  - Artificial Intelligence
  - Philosophy and cognitive-science.

# Unified Modelling Language (UML)

# UML History

- OO languages appear mid 70's to late 80's (cf. Budd: communication and complexity)
  - Between '89 and '94, OO methods increased from 10 to 50.
  - Unification of ideas began in mid 90's.
    - Rumbaugh joins Booch at Rational '94
    - v0.8 draft Unified Method '95
      - Jacobson joins Rational '95
    - UML v0.9 in June '96
    - UML 1.0 offered to OMG in January '97
    - UML 1.1 offered to OMG in July '97
      - Maintenance through OMG RTF
    - UML 1.2 in June '98
    - UML 1.3 in fall '99
    - UML 1.5 <http://www.omg.org/technology/documents/formal/uml.htm>
    - UML 2.0 underway <http://www.uml.org/>
  - IBM-Rational now has *Three Amigos*
    - Grady Booch - Fusion
    - James Rumbaugh – Object Modeling Technique (OMT)
    - Ivar Jacobson – Object-oriented Software Engineering: A Use Case Approach (Objectory)  
( And David Harel - StateChart)
  - Rational Rose <http://www-306.ibm.com/software/rational/>
- 
- The timeline is divided into three main phases:
- pre-UML**: This phase covers the period from 1994 to 1996. It includes the joining of Rumbaugh and Booch at Rational, the creation of the v0.8 draft Unified Method, and the release of UML v0.9 in June 1996.
  - UML 1.x**: This phase covers the period from 1997 to 1999. It includes the offering of UML 1.0 to OMG in January 1997, the offering of UML 1.1 to OMG in July 1997 (with maintenance through OMG RTF), the release of UML 1.2 in June 1998, the release of UML 1.3 in fall 1999, and the release of UML 1.5.
  - UML 2.0**: This phase covers the period from 2000 to 2002. It includes the underway status of UML 2.0.

# Unified Modeling Language (UML)

- An effort by IBM (Rational) – OMG to standardize OOA&D notation
- Combine the best of the best from
  - Data Modeling (Entity Relationship Diagrams);  
Business Modeling (work flow); Object Modeling
  - Component Modeling (development and reuse - middleware, COTS/GOTS/OSS/...:)
- Offers vocabulary and rules for **communication**
- ***Not*** a process but a language

*de facto* industry standard

# Unified Modeling Language

---

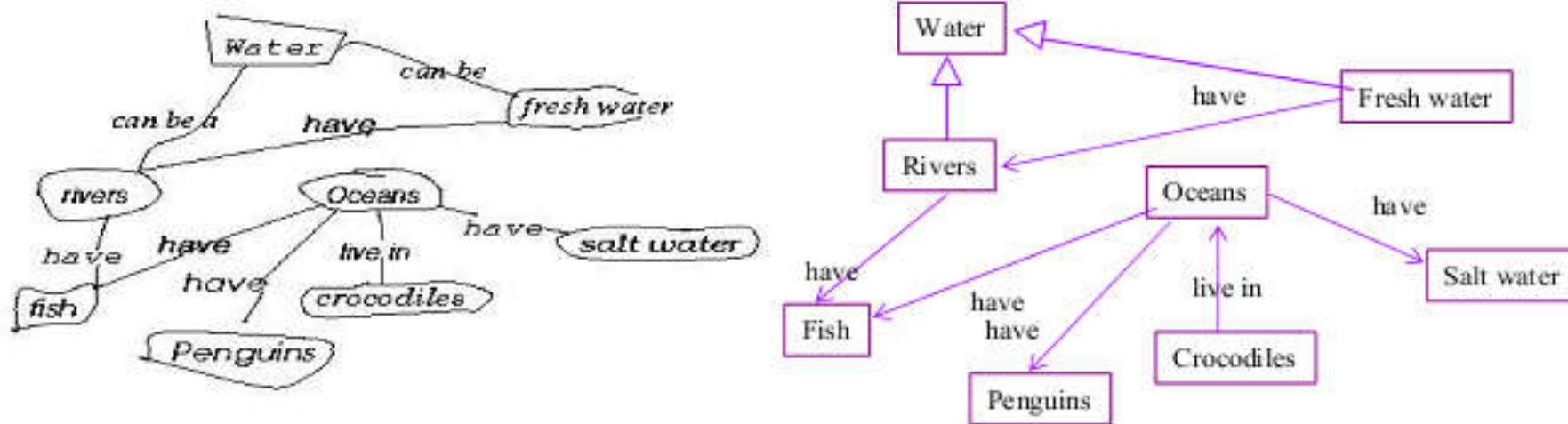
- General purpose modelling language
  - Aim is to define standard way to visualize
    - How a system is developed
    - How the system would perform
    - Who are the stakeholders of the system
    - What would be the interactions between the stakeholders
    - Etc.
-

# UML is for Visual Modeling



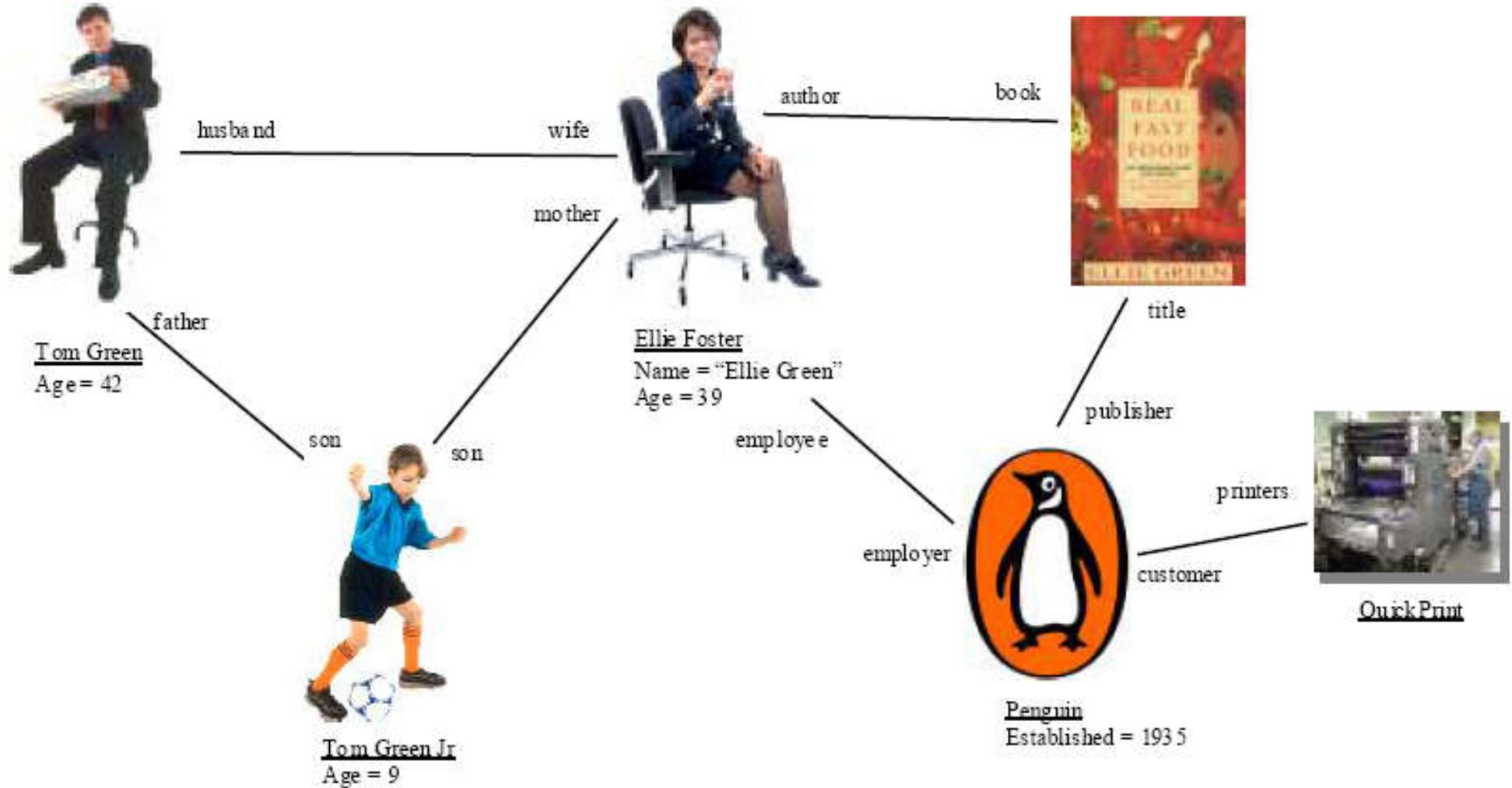
*A picture is worth a thousand words!*

# Three (3) basic BUILDING blocks of UML



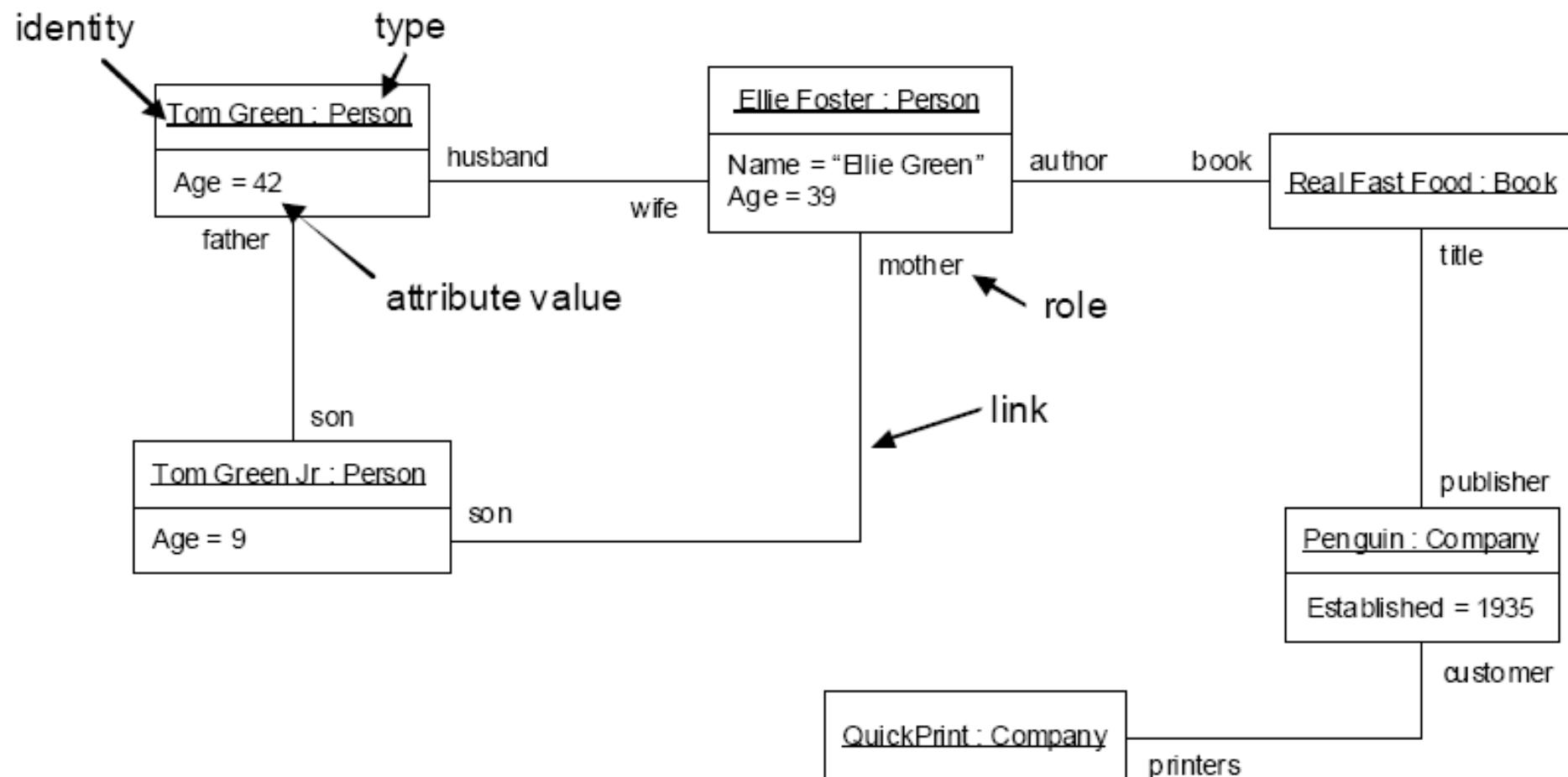
- Things - important modeling concepts
  - Relationships - tying individual things
  - Diagrams - grouping interrelated collections of things and relationships
- Just glance thru  
for now*

# Relationships between Objects/Entities at some point



# Object Diagrams

More formally in UML



# Model types

---

- **Functional model:** What are the functions of the system?
  - **Object model:** What is the structure of the system? What are the objects and how are they related?
  - **Dynamic model:** How does the system react to external events? How does the events flow in the system ?
-

# UML Basic Notation Summary

- UML provides a wide variety of notations for modeling many aspects of software systems
- Example diagrams with respect to the different Models in UML:
  - Functional model: Use case diagram
  - Object model: Class diagram
  - Dynamic model: Sequence diagrams, state chart

# Unified Modeling Language (UML) - Diagrams

A connected graph: Vertices are things; Arcs are relationships/behaviors.

## UML 1.x: 9 diagram types.

### Structural Diagrams

Represent the *static* aspects of a system.

- Class;
- Object
- Component
- Deployment

### Behavioral Diagrams

Represent the *dynamic* aspects.

- Use case
- Sequence;
- Collaboration
- Statechart
- Activity

## UML 2.0: 12 diagram types

### Structural Diagrams

- Class;
- Object
- Component
- Deployment
- Composite Structure
- Package

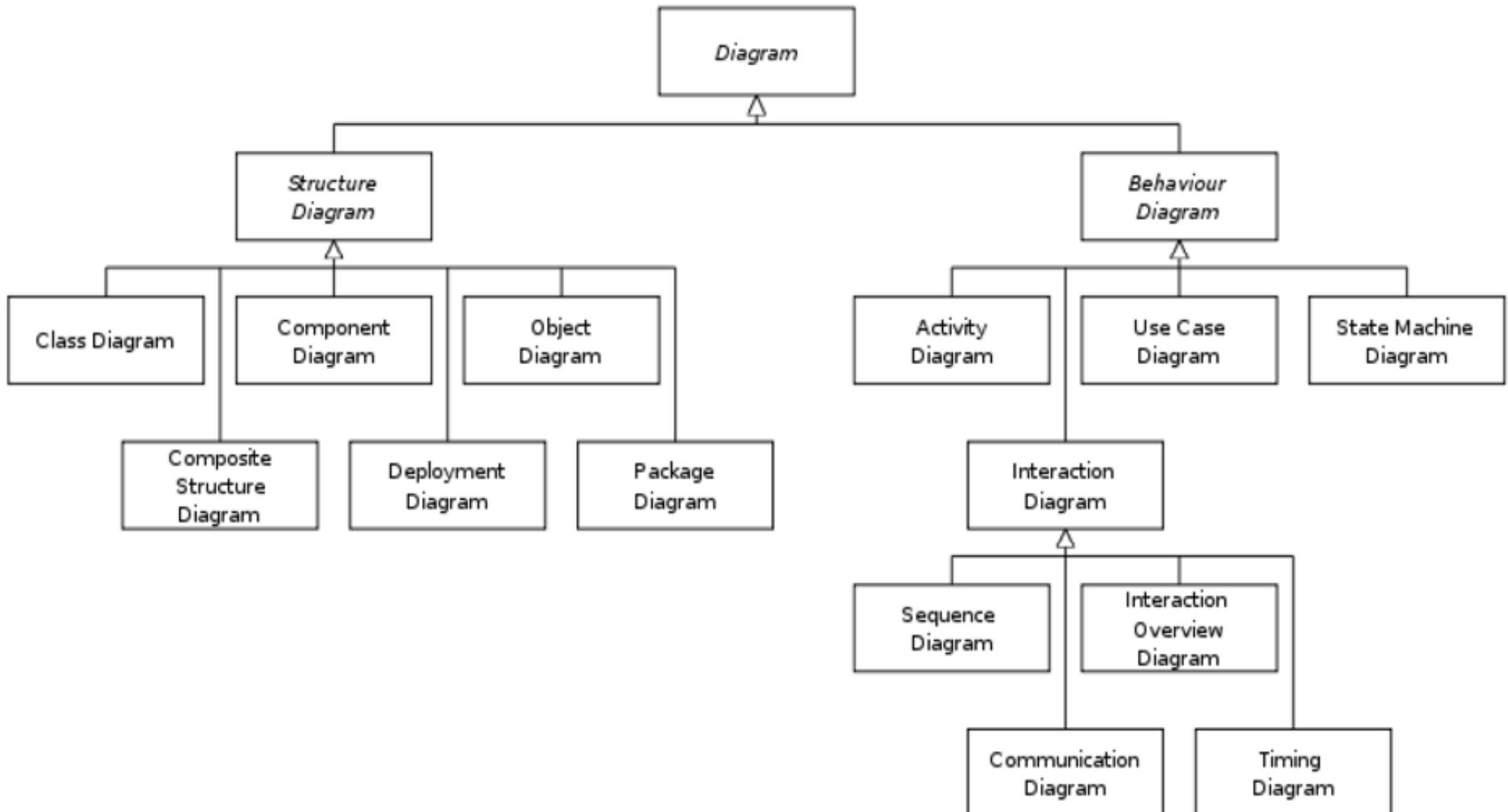
### Behavioral Diagrams

- Use case
- Statechart
- Activity

### Interaction Diagrams

- Sequence;
- Communication*
- Interaction Overview
- Timing

# Unified Modeling Language (UML) - Diagrams



# UML Core Conventions

- All UML Diagrams denote graphs of nodes and edges
  - Nodes are entities and drawn as rectangles or ovals
  - Rectangles denote classes or instances
  - Ovals denote functions
- Names of Classes are not underlined
  - SimpleWatch
  - Firefighter
- Names of Instances are underlined
  - myWatch:SimpleWatch
  - Joe:Firefighter
- An edge between two nodes denotes a relationship between the corresponding entities

# What is a Use Case?

- Created by Ivar Jacobson (1992)
- “A use case is a sequence of transactions in a system whose task is to yield a measurable value to an individual actor of the system”
- **Describes WHAT the system (as a “Black Box”) does from a user’s (actor) perspective**
- **The Use Case Model is NOT an inherently object oriented modeling technique.**

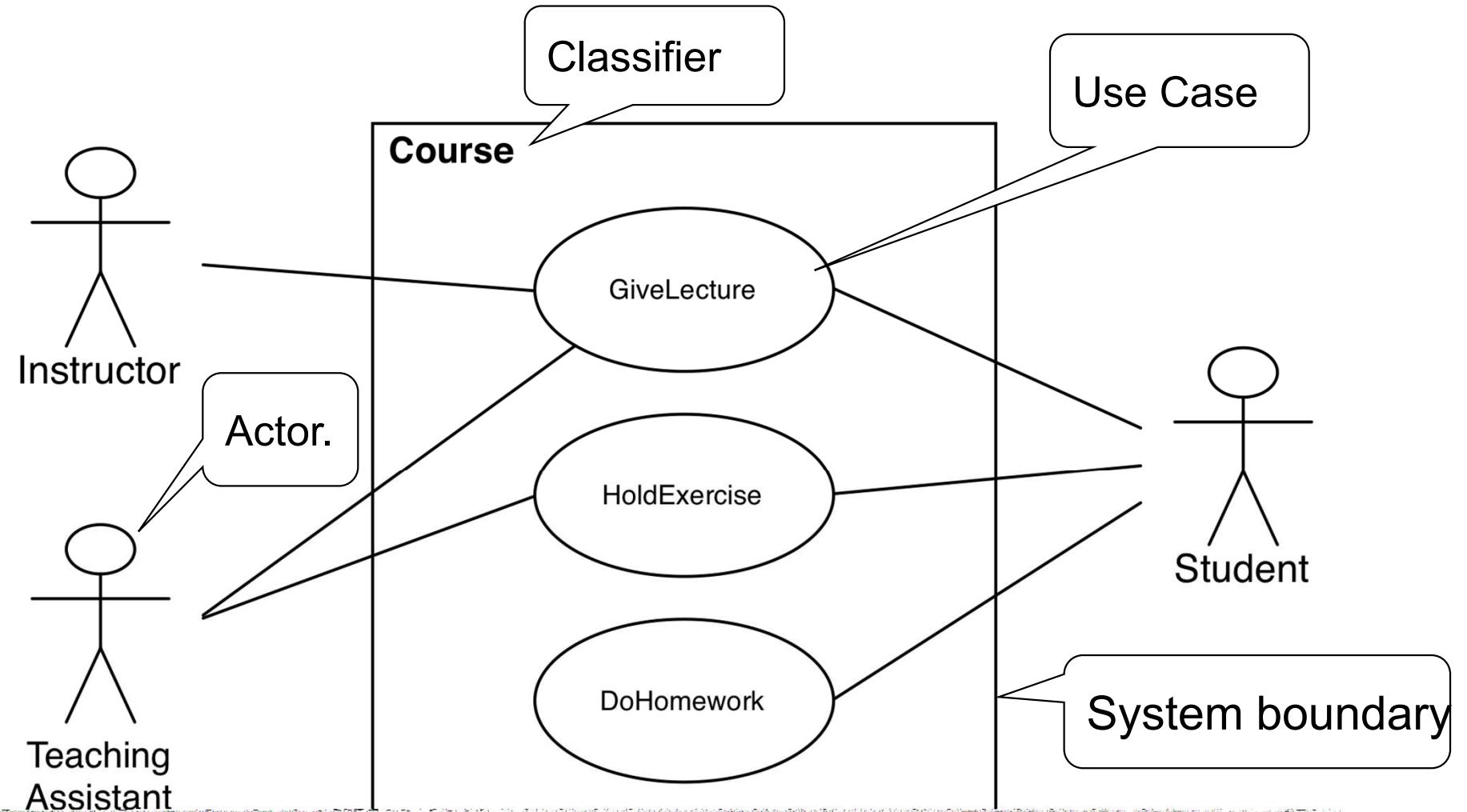
# Benefits of Use Cases

- Captures operational requirements from user's perspective
- Gives a clear and consistent description of what the system should do
- A basis for performing system tests
- Provides the ability to trace functional requirements into actual classes and operations in the system

# UML Use Case Diagrams

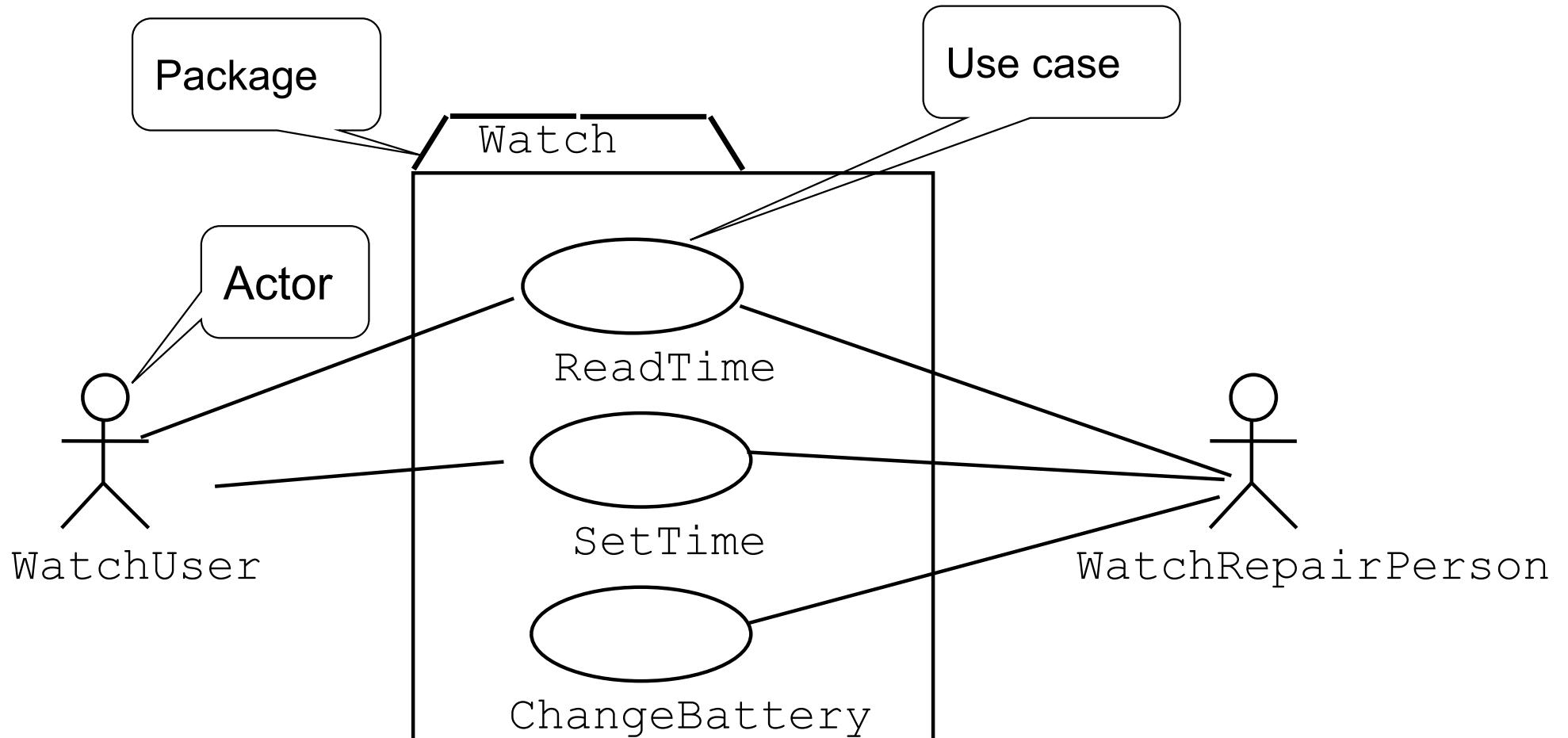
- A Use Case model is described in UML (Unified Modeling Language) as one or more Use Case Diagrams (UCDs)
- A UCD has 4 major elements:
  - The **system** described
  - The **actors** that the system interacts with
  - The **use-cases**, or services, that the system knows how to perform
  - The **relationships** between the above elements.

# UML first pass: Use case diagrams



Use case diagrams represent the functionality of the system from user's point of view. Use cases are the tasks that a user would perform.

# Historical Remark: UML 1 used packages



Use case diagrams represent the functionality of the system from user's point of view

# System

- As part of use-case modeling, the **boundaries of the system** developed must be defined
- Defining the boundaries of the system is not trivial
  - Which tasks are automated and which are manual?
  - Which tasks are performed by other systems?
    - The entire solution that we supply should be included in the system boundaries
    - Incremental releases

# System (cont.)

- A system in a UCD is represented as a box
- The name of the system appears above or inside the box

Traffic Violations Report System

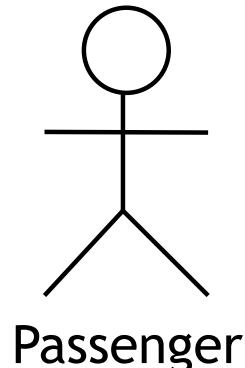
# Actor

- Someone or something that interacts with the system (exchanges information with the system)
- An actor represents a role played with respect to the system, not an individual user of the system
- Example:
  - Policeman – Enters data
  - Supervisor – Allowed to modify/erase data
  - Manager – Allowed to view statistics
- A single actor may play more than one role.

# Actor (cont.)

- Actors have **goals**:
  - Add a Traffic Violation
  - Lookup a Traffic Violation
- **Actors don't need to be human**
  - May be an external system that interfaces with the developed system
- An actor has a name that reflects its role.

# Actor Icons



- An actor models an external entity which communicates with the system:
  - User
  - External system
  - Physical environment
- An actor has a unique name and an optional description
- Examples:
  - Passenger: A person in the train
  - GPS satellite: Provides the system with GPS coordinates

# Use Case



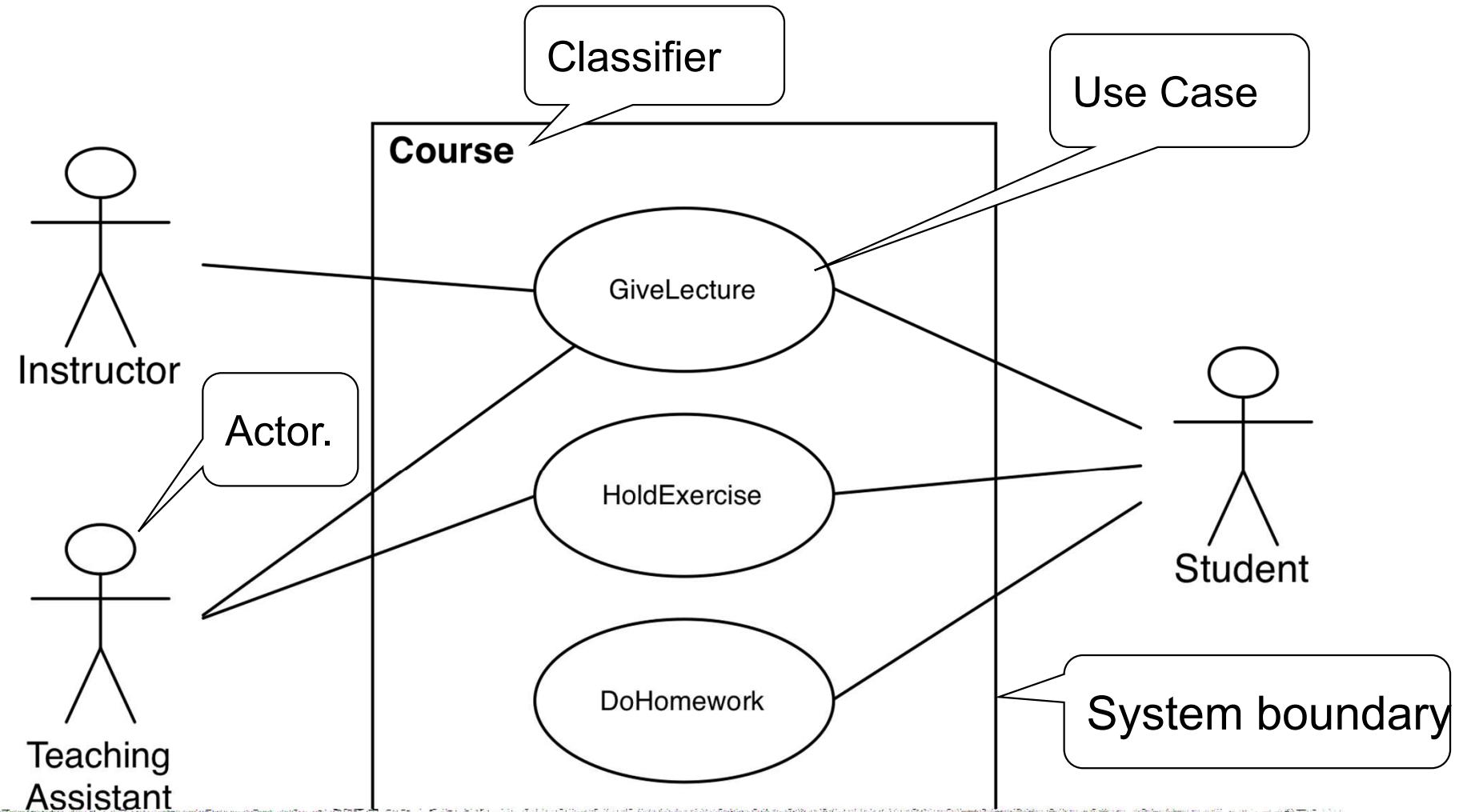
- A use case achieves a goal of value to an actor
- System does these things for actors
- Describes what system is for?  
And Not about how it does it
- Starts with an active verb from the point of view of the system

# Communications



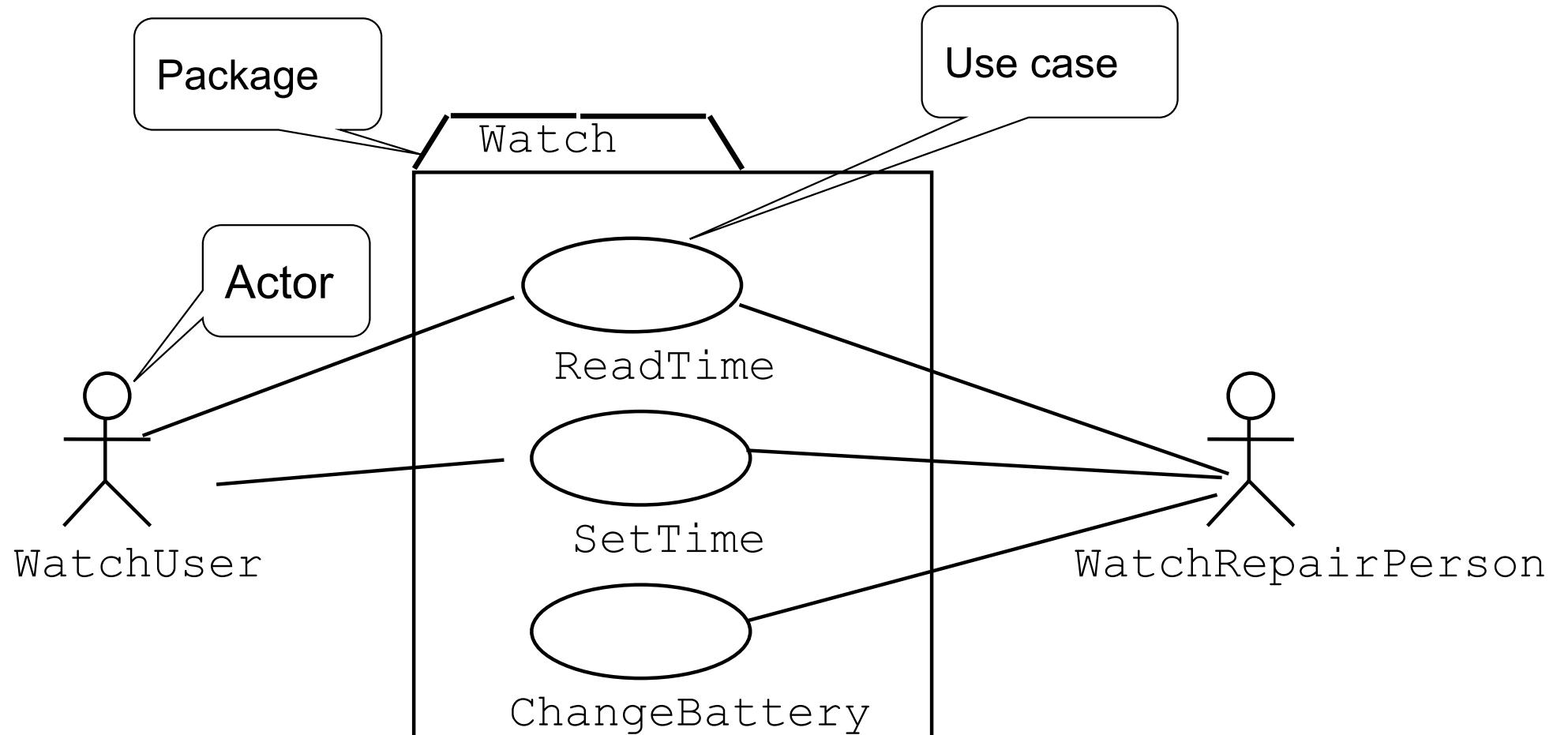
- Lines between Actor and Use Case summarize interactions graphically
- Actors and use cases exchange information to achieve the goal but the details of interaction are not shown

# UML first pass: Use case diagrams



Use case diagrams represent the functionality of the system from user's point of view. Use cases are the tasks that a user would perform.

# UML first pass: Use case diagrams



Use case diagrams represent the functionality of the system from user's point of view

# Diagrams in UML – Actors in Use case

---

- There is no standard available to choose Actors for a system
  - Selection of Actors and Use cases of a system solely depends upon the development team.
-

# Diagrams in UML

The UTD wants to computerize its registration system

- The Registrar sets up the curriculum for a semester
- Students select 3 core courses and 2 electives
- Once a student registers for a semester, the billing system is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- Professors use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation

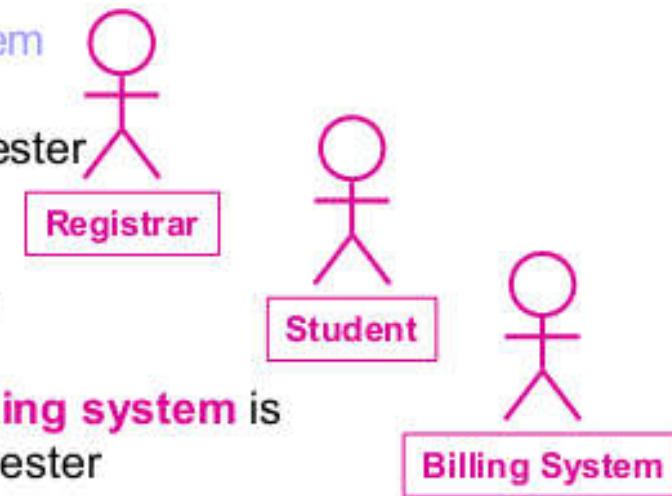
***What's most important?***

# Diagrams in UML – Actors in Use case

- An **actor** is someone or some thing that must interact with the system under development

The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- **Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- **Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register
- **Users** of the registration system are assigned passwords which are used at logon validation

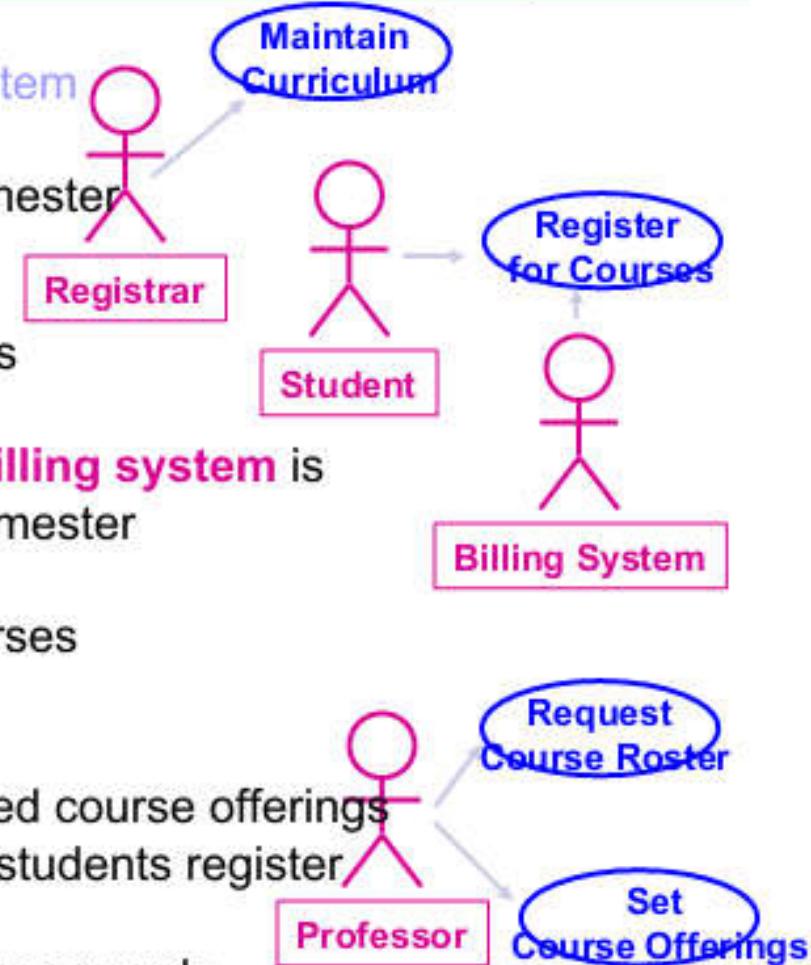


# Diagrams in UML – Use cases in Use case diagram

- A **use case** is a sequence of interactions between an actor and the system

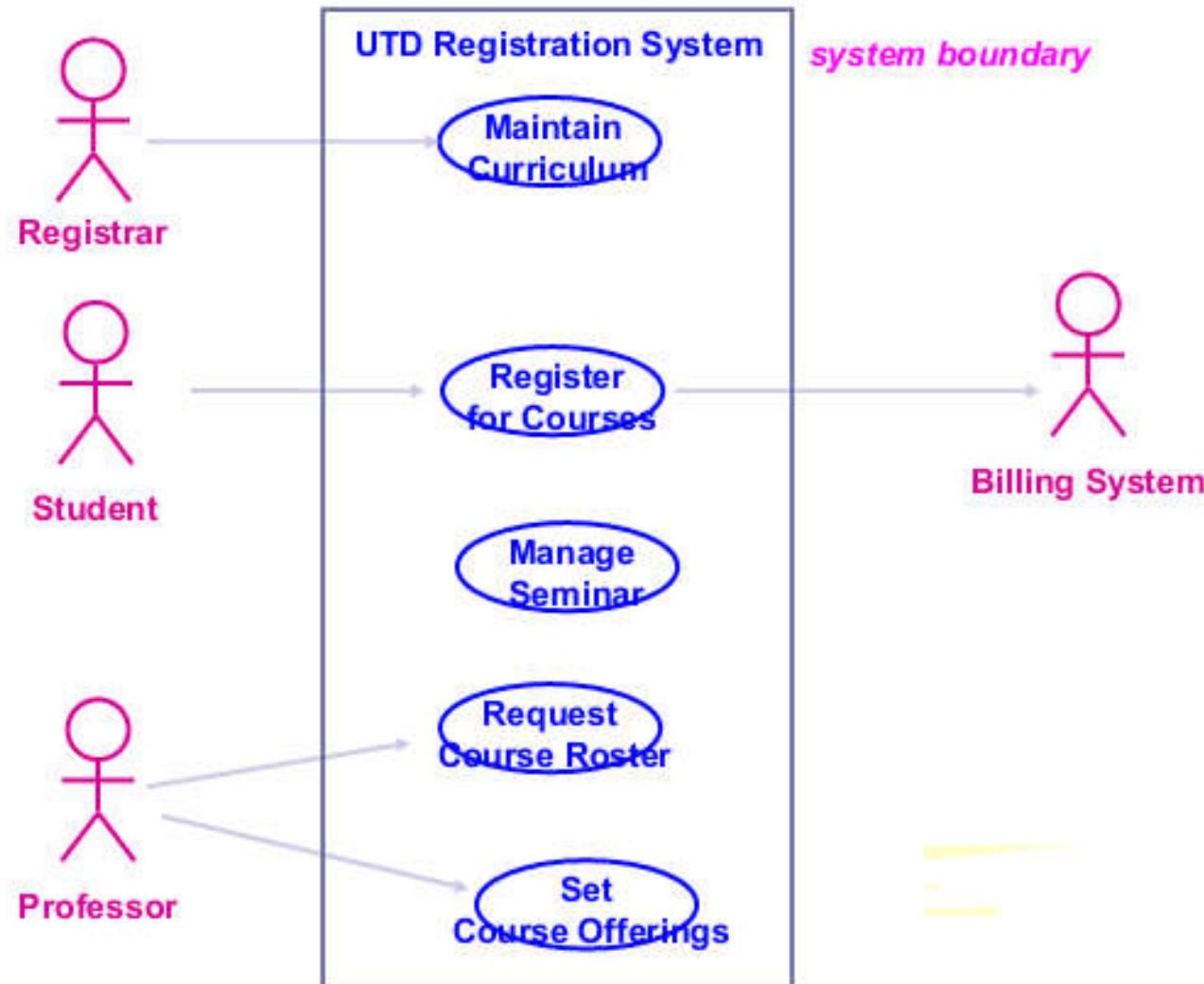
The UTD wants to computerize its registration system

- The **Registrar** sets up the curriculum for a semester
- **Students** select 3 core courses and 2 electives
- Once a student registers for a semester, the **billing system** is notified so the student may be billed for the semester
- Students may use the system to add/drop courses for a period of time after registration
- **Professors** use the system to set their preferred course offerings and receive their course offering rosters after students register
- Users of the registration system are assigned passwords which are used at logon validation



# Diagrams in UML – Use case Diagram

- Use case diagrams depict the relationships between actors and use cases

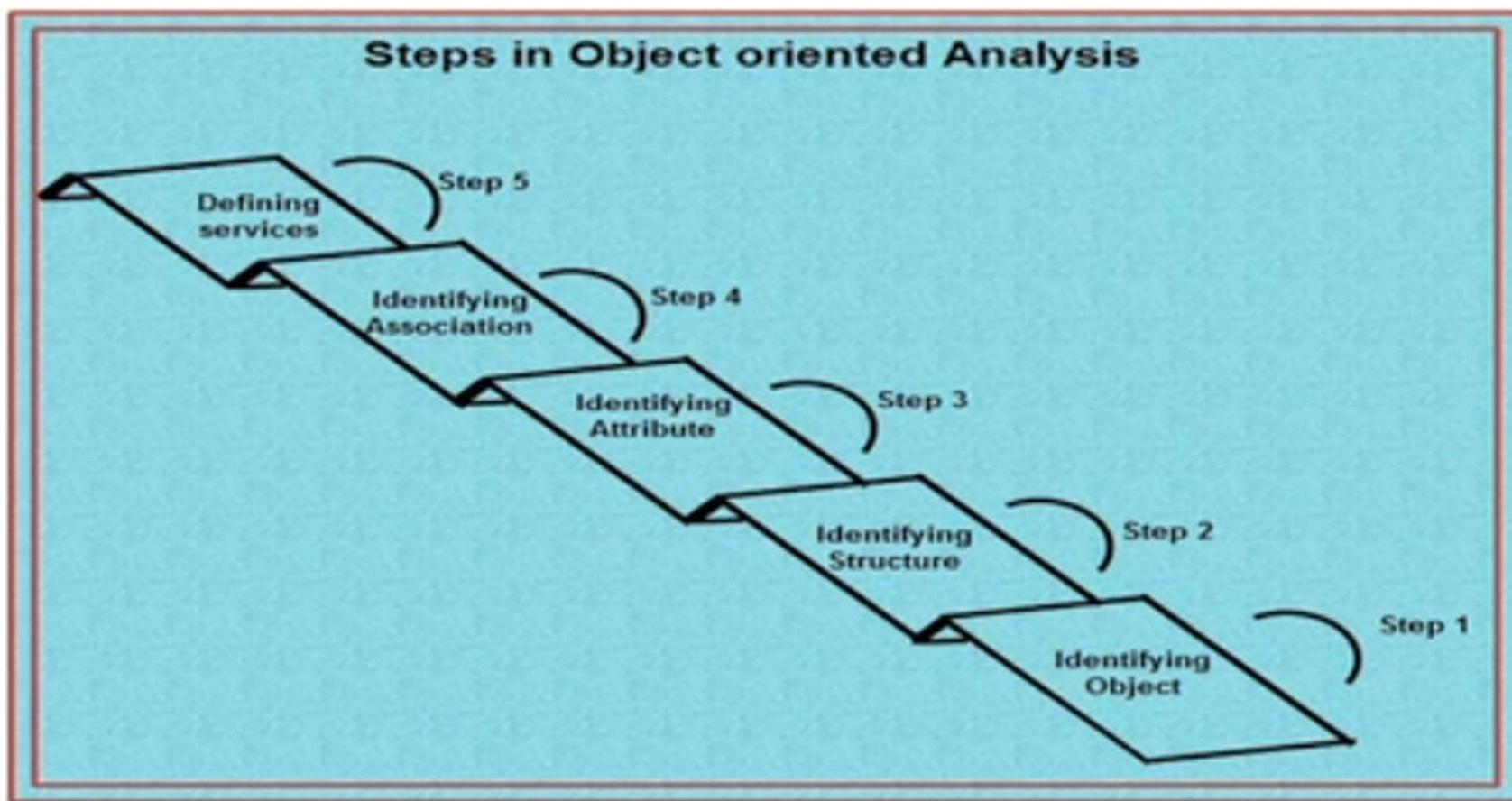


# OOA – OOD - OOP

- Object Oriented Analysis (OOA)
- Object Oriented Design (OOD)
- Object Oriented Programming (OOP)
- Interrelation between OOA, OOD, and OOP

# Object Oriented Analysis (OOA)

- Object Oriented Analysis (OOA) is a method of analysis that examines **requirements** from the perspective of the **classes** and **objects** found in the **vocabulary of the problem domain**.

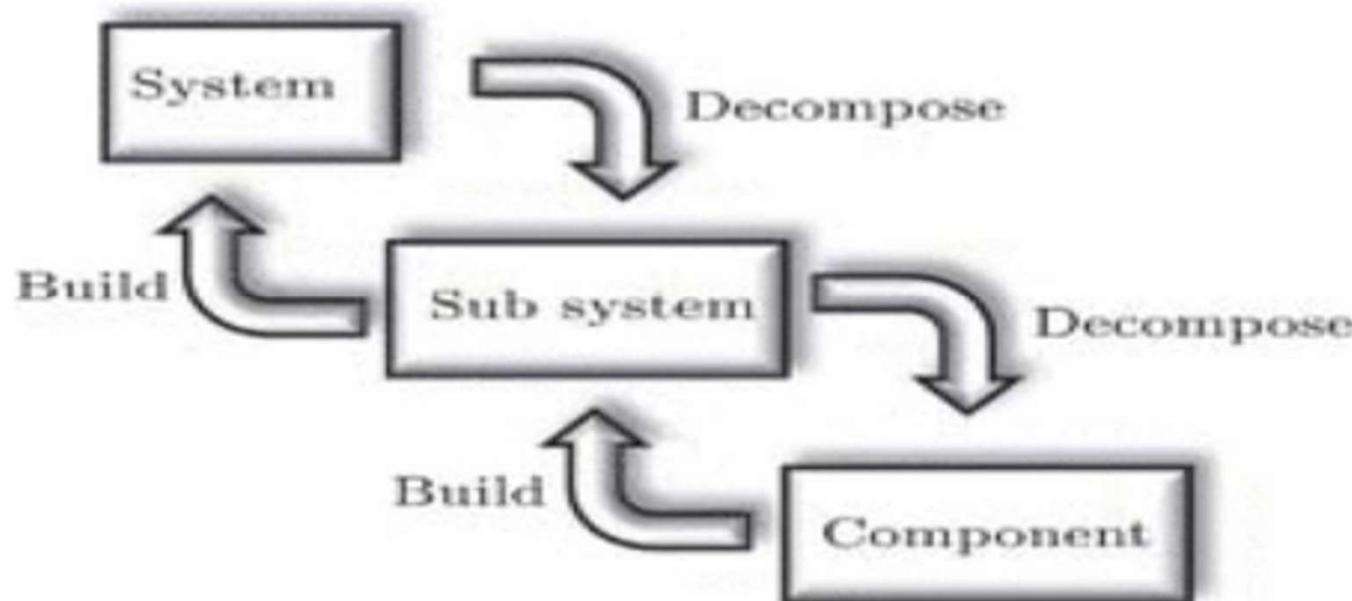


# Object Oriented Analysis (OOA)

- Stages for Object Oriented Analysis (OOA)
  - Identifying Objects
  - Identifying Structure
  - Identifying Attributes
  - Identifying Associations
  - Defining services

# Object Oriented Design (OOD)

- It follows the Object Oriented Decomposition approach
- Three stages in the designing process
  - Logical models (class and object structure)
  - Physical models (module and process architecture)
  - Behavioral models (statics and dynamics of the system)



# Object Oriented Design (OOD)

- Stages for Object Oriented Design (OOD)
  - Defining the context of the system
    - Usability of the system
    - Number of users
  - Designing system architecture
    - Dependencies / associations between components
    - Centralized / Decentralized ?
    - Database schema required
  - Identification of objects in the system
    - Entities involved
  - Construction of design models
    - Types of models used for designing the system
  - Specification of object interfaces
    - Depends upon the objects and architecture of the system.

# Object Oriented Design (OOD)

- Object design includes the following phases (broadly)
  - Identify the objects
  - Representation of the objects (design models)
  - Classification of operations
    - Who is going to use?
    - What are the restrictions?
  - Algorithm design
    - At what point during the design phase we should think about the algorithm which would be running in the system?
  - Association and Relationship among entities
  - Identifying the external interactions and including them in the design process
  - Package classes and associations into modules.

# Object Oriented Programming (OOP)

- Analysis is the extraction of objects from the concepts and notions
- Analysis gives us certain set of objects and their inherent properties
- Analysis also gives us how an object performs (theoretically)
- What messages need to be sent or received by the objects are also included in the analysis phase
- OOD gives us the opportunity to put the concepts learnt from OOA into the system
- OOD deals with the system architecture and ways in which the observations from OOA can be induced into the system
- OOD also deals with handling the limitations of the system while inducing the observations from OOA.

# Object Oriented Programming (OOP)

- Now using the inputs from OOA and OOD subsequently we need to build the real-life system
- The real-life system would be the final finished product
- We need programming languages for implementation and a successful running of the designed system
- Choosing the programming language for the designed system is very important .. It depends upon specification and requirements in the design.. And also on the features of the programming language
- If we observe that the developed application needs to be moved across different platforms (system and server configuration).. Then one may use Python/Java which is very portable.

# Object Oriented Programming (OOP)

- On the other hand, Python is extremely slow in processing
- Therefore if there is a time constraint or processing power constraint then we may choose a programming language which can be executed very fast (C/C++)
- Choice of programming language selection would depend on how the analysis is being done and how the system components are designed
- **Choice of OOP language does not depend upon popularity of the language**
  - It depends upon
    - What we need to do
    - What are the constraints in the system we are developing
    - What are the features of the chosen OOP language.

# Object Oriented Programming (OOP)

- On the other hand, Python is extremely slow in processing
- Therefore if there is a time constraint or processing power constraint then we may choose a programming language which can be executed very fast (C/C++)
- Choice of programming language selection would depend on how the analysis is being done and how the system components are designed
- Choice of OOP language does not depend upon popularity of the language
- It depends upon
  - What we need to do
  - What are the constraints in the system we are developing
  - What are the features of the chosen OOP language
- OOP is a programming language model organized around **objects** rather than actions and data.

# Object Oriented Programming (OOP)

## ■ **Re-usability**

- Reusing some facilities rather than building them again and again
- It is achieved by using Class concept
- Any number of times we can use the contents of a class using entities such as Objects

## ■ **Data redundancy**

- Same piece of data is held in two separate places
- If a user wants a similar functionality in multiple classes
- Then we can write common class definitions for similar functionalities and inherit them

# Object Oriented Programming (OOP)

## ■ **Code maintenance**

- Maintenance of software includes fixing of bugs, improving the code quality, etc.
- A well structured code where Object oriented concepts are properly applied is easier to be maintained by developers
- Even very helpful to new employee who is "inheriting" the existing project as time passes

## ■ **Security**

- Object oriented concepts such as Abstraction help in hiding data
- Developers have option to conditionally expose data to externals.

# Object Oriented Programming (OOP)

## ■ Design benefits

- The design benefits are in terms of designing, fixing things easily and eliminating risks
- Developers have the option to create a better design of the software with fewer flaws

## ■ Better productivity

- Using the above features the final product is improved
- More features are included. Final product is easier to read, use, and maintain
- Programmer can easily add new features to existing product.

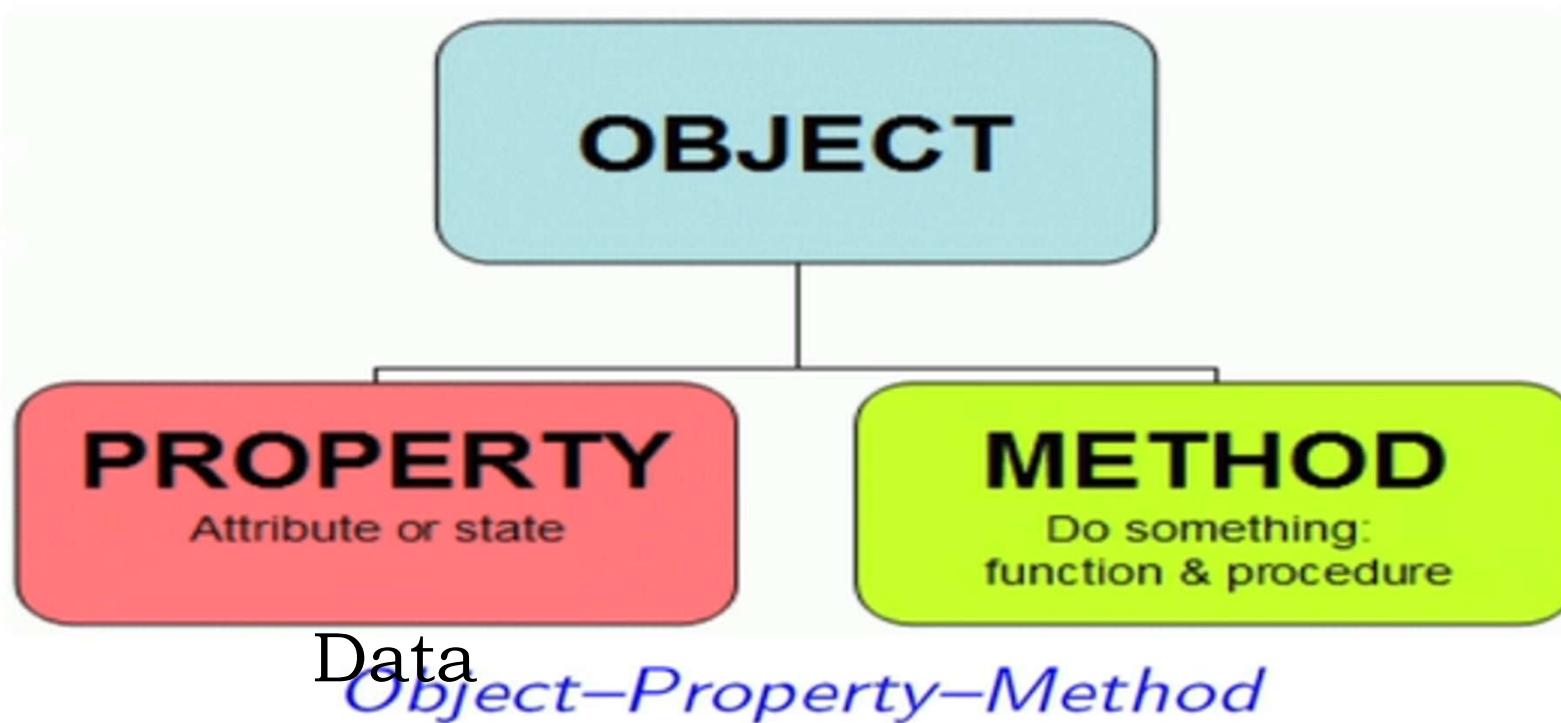
# Class - Object

- Abstraction hierarchy gives Class structure
- How many classes?
- How many objects?
- Object of which classes?
- Why we create object of those classes?
  
- *Vehicle->[**4whl**, 3whl, 2whl]->[Car, Truck, Bus]->[Sedan, SUV, HatchBack]->[Tigor, Nexon, Tiago]*
  
- Why Tiago would have all property of Cars?
- Why would we reveal all property of Cars to Tiago?
- Tiago should have property of Hatchback and nothing else
- Tiago should not know task of Tigor
- Tiago would know property that are specific to Hatchback and not inherit properties from Car class
- We can do specific changes to Tiago without changing property of Tigor (modularity)

# Object Oriented Programming (OOP)

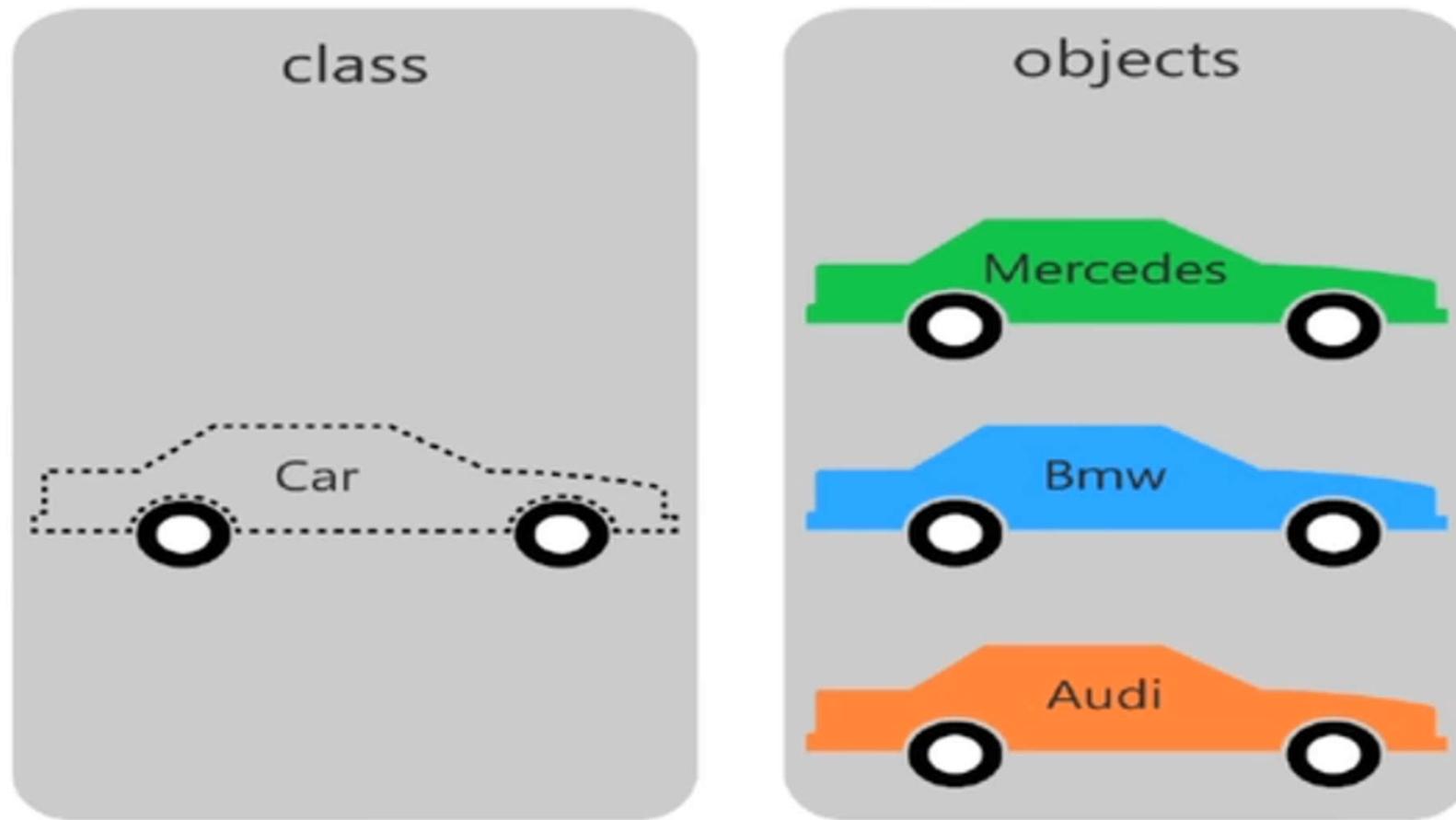
- Three important parts of OOP

- Object
- Class [Type]
- Inheritance



# Class

- Objects have an associated Type which is known as the Class



*Class [type]-Object*

# Class – Objects – Methods

- Object oriented programs deal with Objects
- Objects are certain entities or modules
- That have certain behavior
- Their behavior may change as per requirement
- They hold information used within the program
- They can interact with one another.

# Class – Objects – Methods

- Here we write codes in separate blocks
- Each block is designed to perform separate tasks
- Such blocks are called the Classes
- Classes are the building blocks of OOP concept
- Each block/class stores information within multiple methods
- Methods within classes are used to perform sub-tasks.

# Class – Objects – Methods

Method - 1

Method - 2

**Class 1**

Method - 1

**Class 2**

Method - 1

Method – 2

Method - 3

**Class 3**

- **Objects** are the instance of a class or executable copy of the class
- Multiple objects might exist for a class
- Each object can perform separate tasks.

# Class

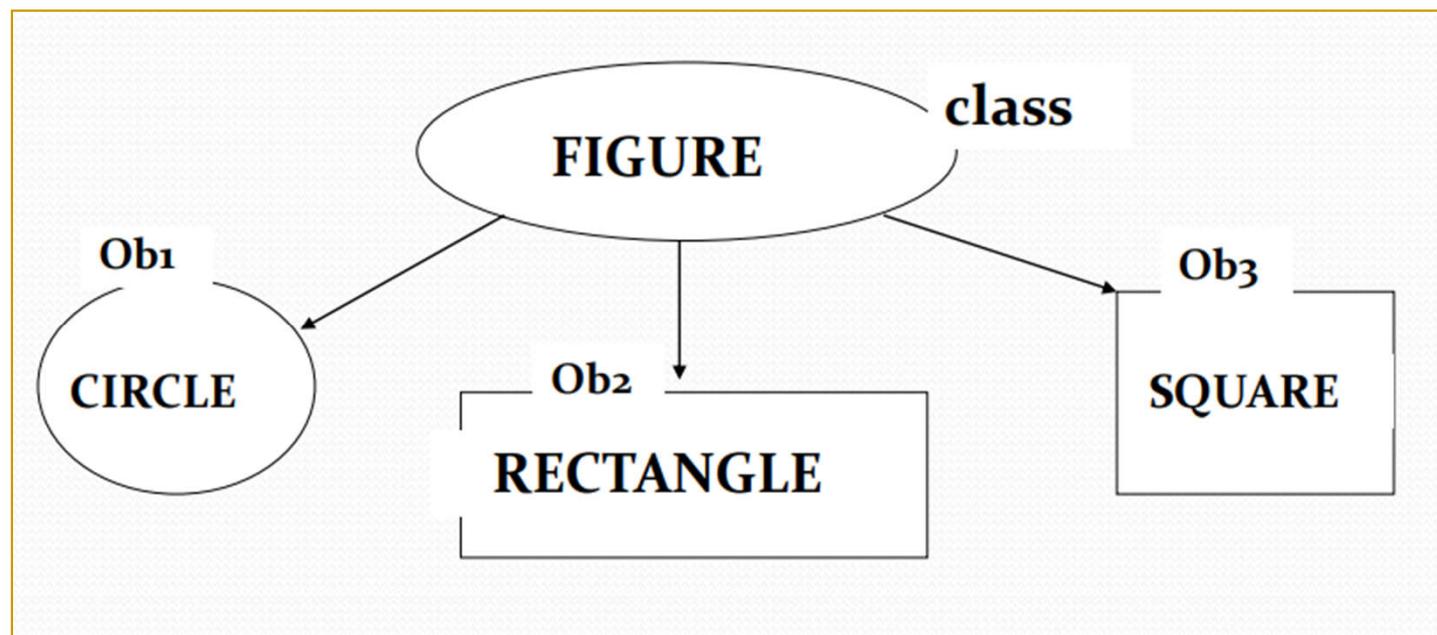
- Class is a structure that defines the data and methods to work on
- Each class has a well-defined responsibility
- The responsibility of a Class is on a higher level
- It consists of instance variables and methods
- It provides modularity and structure in an object-oriented computer program.

# Methods

- Class consists of collection of methods
- Methods define specific sub-tasks that are assigned for a class
- Communication between classes are performed using methods
- Methods within a class is accessed using Objects.

# Objects

- Objects are instances of a class or an executable copy of a class
- There can be multiple objects of a class



# First program

**class test**

```
{  
    public static void main (String args[ ])  
    {  
        System.out.println ("Hello World!!");  
    }  
}
```

## ■ **class test**

- Declares a class named as test
- Contains a single method “main( )”
- “class” is a keyword
- “test” is the identifier which provides name to the new class.

# First program

- **public static void main(String args[ ])**

- **public**

- It is an Access specifier
  - It is used to declare the method main( ) as unprotected
  - Makes it accessible to all other classes

- **static**

- Every method is accessed using the object of a particular class
  - Static method can be accessed directly without using objects
  - main( ) should be declared static as it is the first method from which execution starts
  - Object creation for the main( ) method container class is not possible prior to execution

- **void**

- main( ) do not return any value

- **String args[ ]**

- Handling command line arguments.

# First program

```
class sample
{
    void display( )
    {
        System.out.println("Pramit");
    }
}

class test
{
    public static void main(String args[])
    {
        sample SS = new sample( );
        SS.display( );
    }
}
```

# Object creation

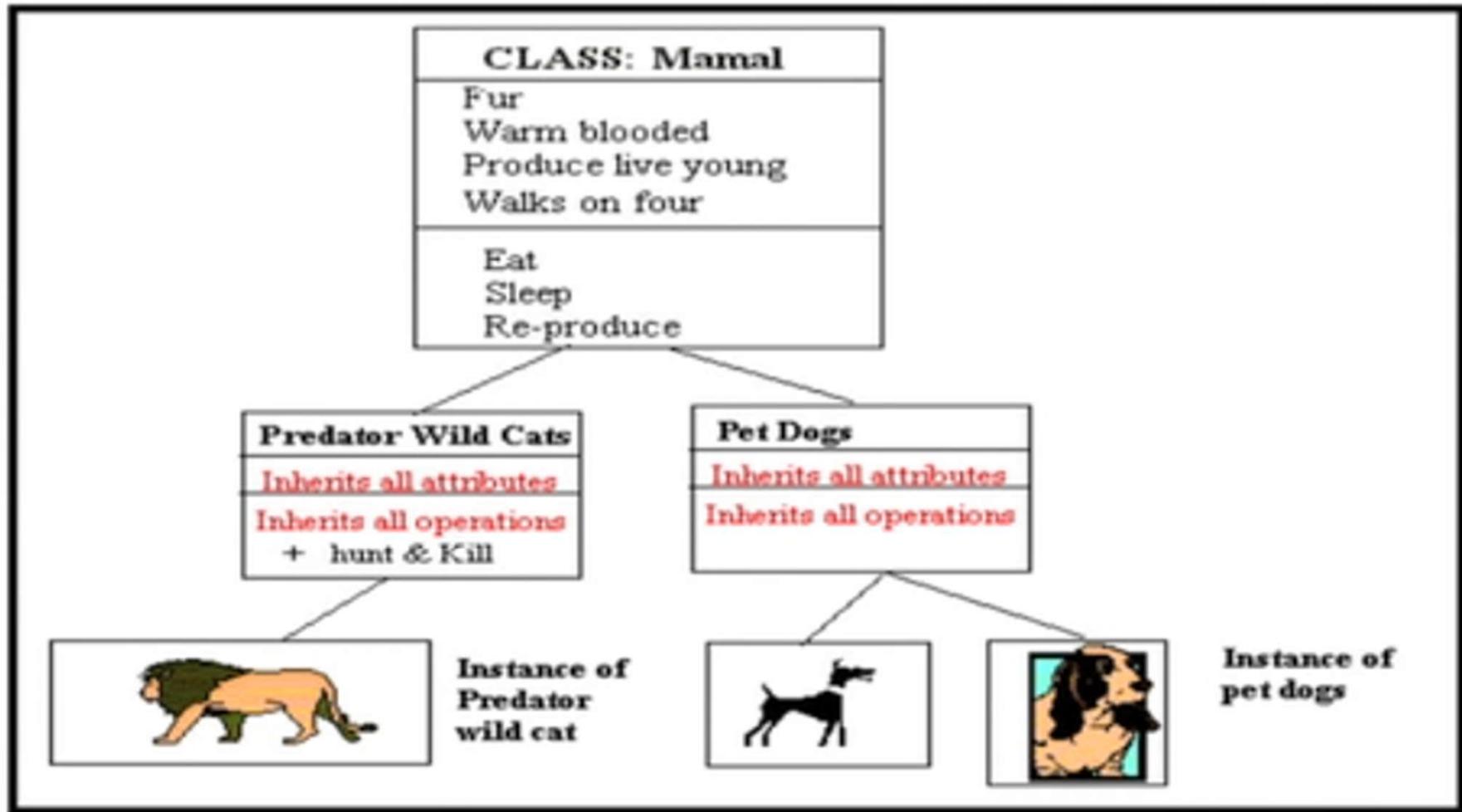
sample SS;

SS = new sample( );

- sample SS
  - Reference created
  - Storage space taken
  - Storage space not yet instantiated
- SS = new sample( )
  - Storage space instantiated
  - Sample class object created.
- Multiple objects can be created to access the same method of a class.

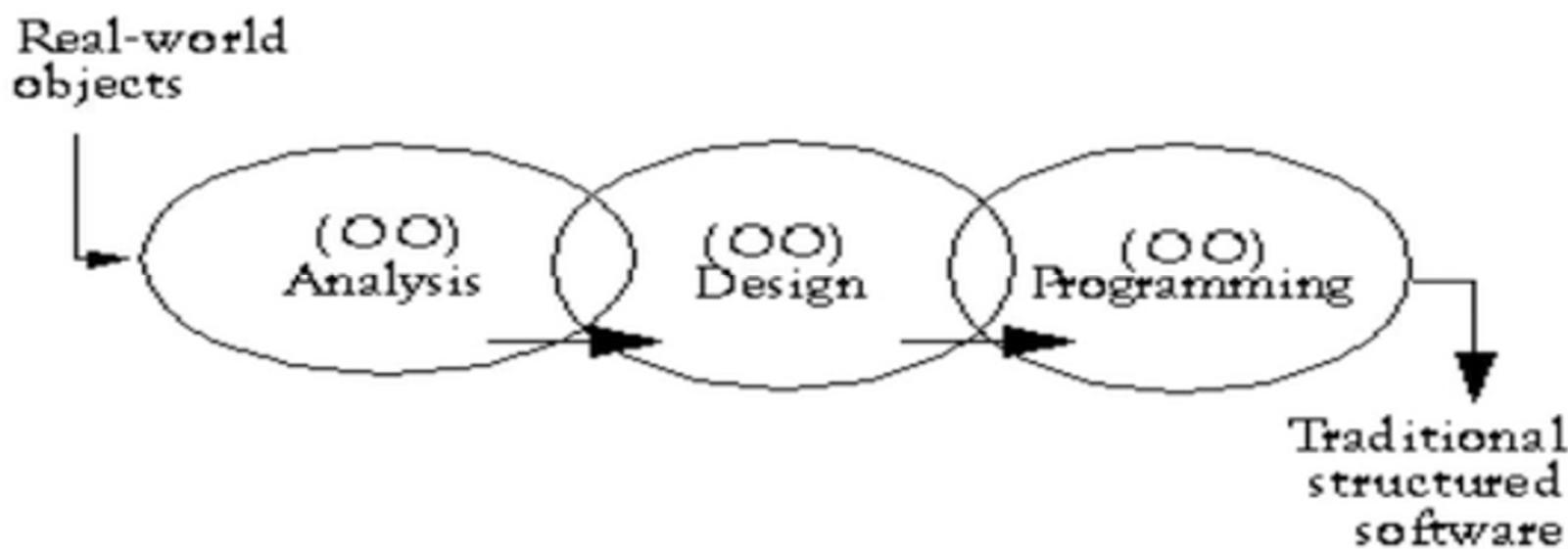
# Inheritance

- Types [classes] may inherit attributes from supertypes [superclasses]



# How are OOA, OOD, and OOP related?

- OOA is concerned with developing an object model that capture the requirements
- OOD is concerned with translating OOA model into a specific model that can be implemented by a software
- OOP is concerned with realizing the OOD model using OOP language such as Java, C++, etc.



# Important Topics

- Understand the major elements / essential characteristics of Object Models
- Understand Abstraction
- Understand Encapsulation

# Elements of Object Models

- Four major elements of the Object Model

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

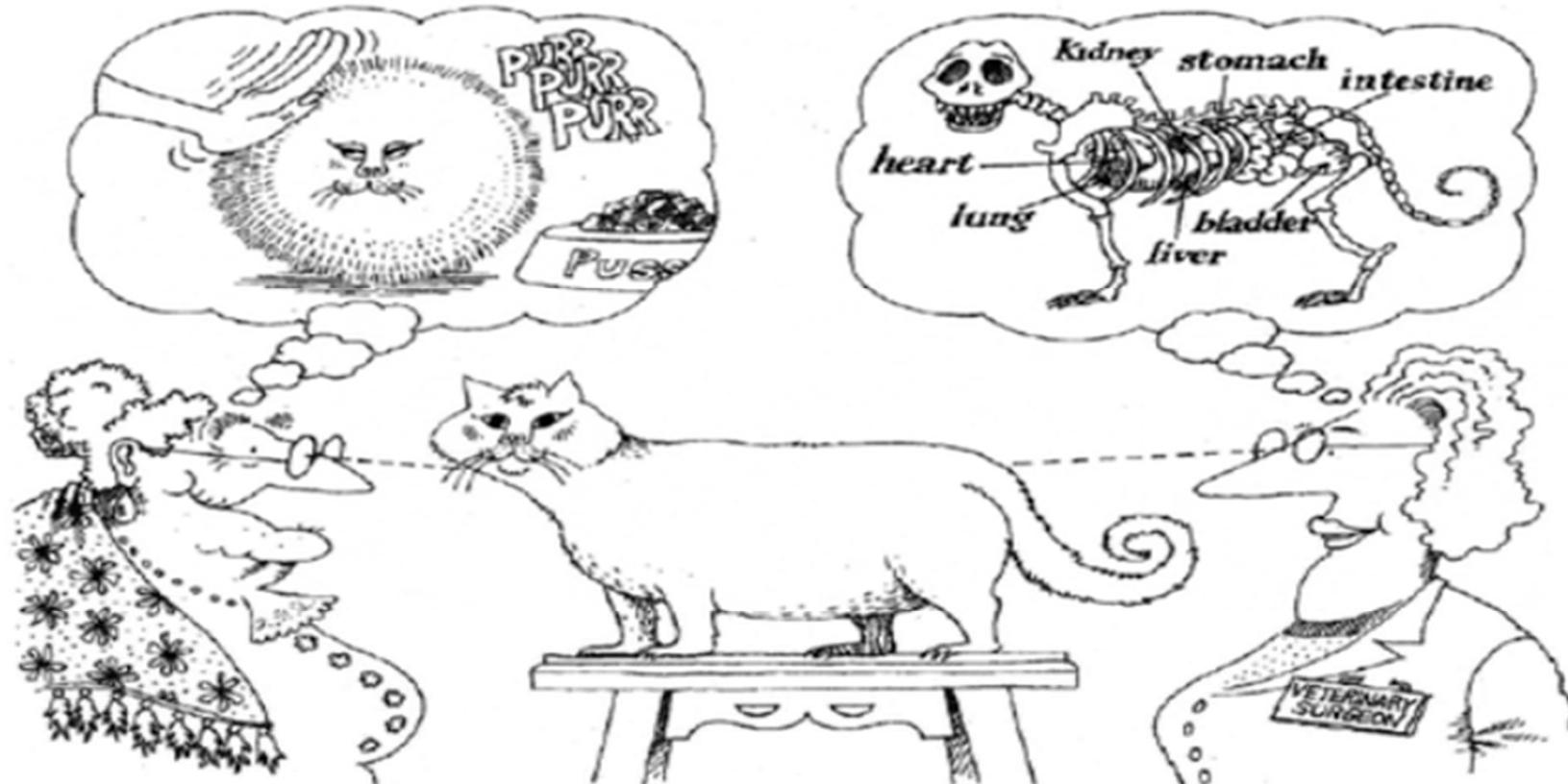
# Elements of Object Models

- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

# Abstraction

- Abstraction is the simplifying a problem by omitting unnecessary details
- Popularly known as the Model Building
- Example:
  - Press the Break to control the Car
  - One knows how to control a Car by pressing Breaks
  - Without knowing how it works !

# Abstraction



*Essential characteristics of an object is relative to the perspective of the viewer*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

Two observers look at the two views of the same object.

# Abstraction

- You are asked to develop an overall understanding of the India
- What would be your approach:
  - Meet all citizens of India?
  - Visit every house?
- You would possibly only refer to various types of Maps for explaining India.

# Abstraction

- Political Map: Administrative structure
- Rivers Map: Natural distribution for water
- Power-Grid Map: Power distribution profile
- Vegetation Map: Cultivation pattern
- Railways Map: Spread of Connectivity
- World Heritage Site Map: Cultural heritage.

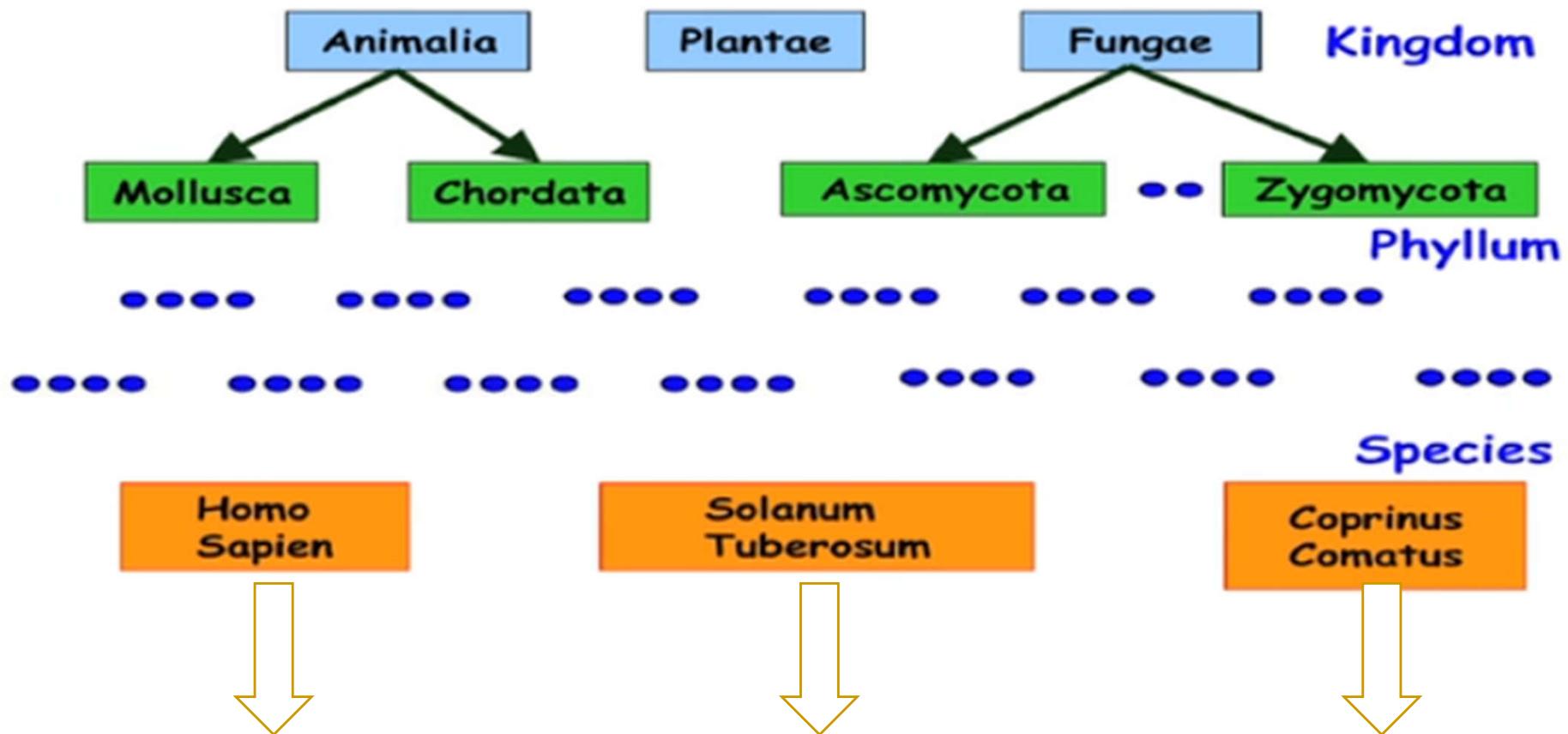
# Abstraction

- Example
- Class-1 is having 10 properties
- Object-1 is created for Class-1
- Object-1 would have all 10 properties
- Now You want to allow User-1 to access the system using Object-1
- Also you want User-1 not to access property 5-10 which are there in Class-1
- You create Class-2 and pass property 5-10 of Class-1 to Class-2
- Thus make Class-2 as generalized class for Class-1 (Inheritance)
- Thus User-1 using Object-1 of Class-1 gets to use all properties without visualizing the working of properties 5-10 which are not written in Class-1.

# Abstraction

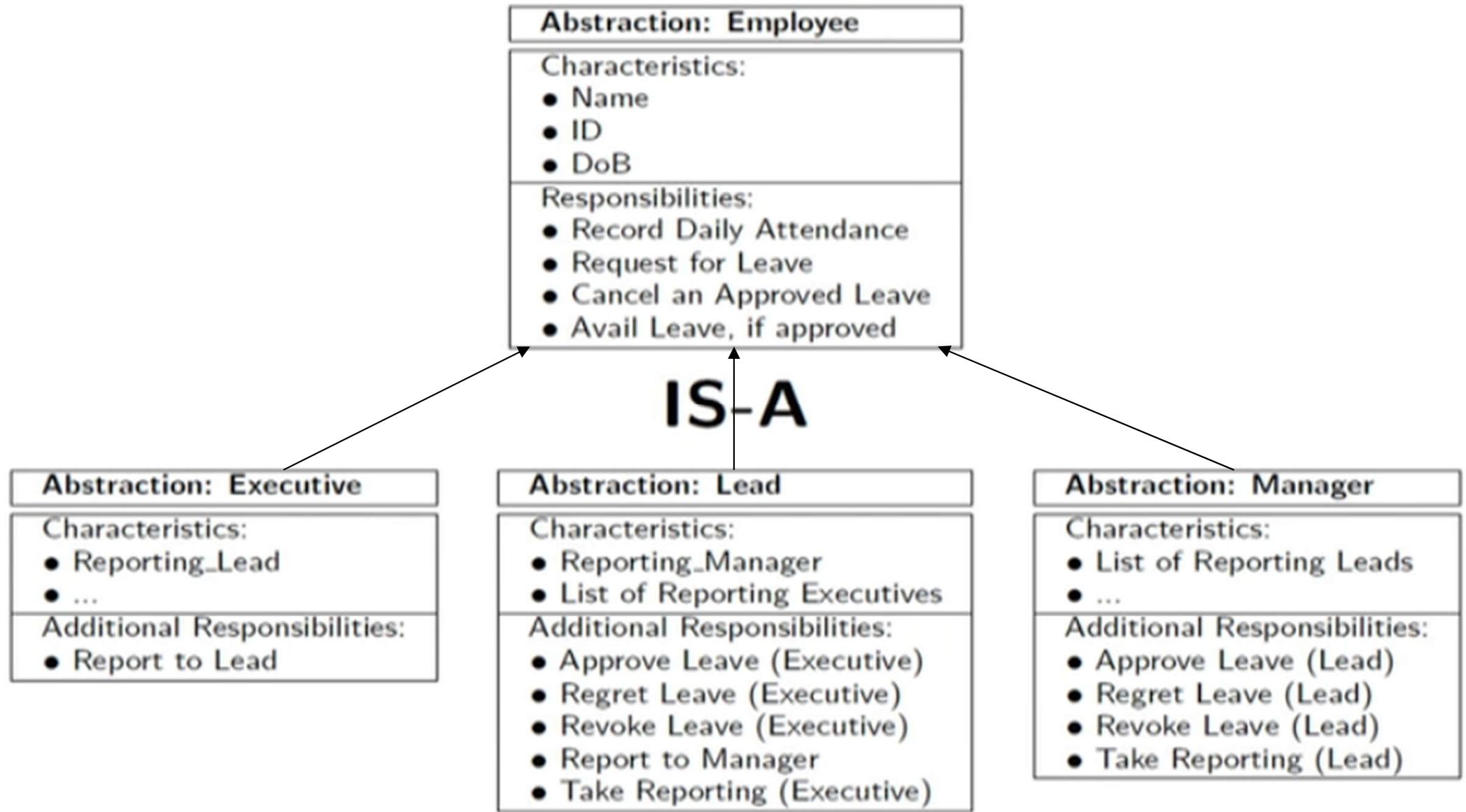
- Several abstractions of the same problem are possible:
  - Each abstraction deals with a specific problem
  - Focus on some specific aspect and ignore the rest
  - Use different models for different aspects of the problem
- For complex problems
  - A single level of abstraction is inadequate
  - Single level does not simplify the system
  - A hierarchy of abstraction needs to be constructed  
(Abstraction Hierarchy)
- Hierarchy of models
  - A model in a layer is abstraction of the lower layer of models
  - Higher layer models implement the model in the layer.

# Abstraction Hierarchy: Living Organisms



More deeper level of hierarchy can be achieved

# Abstraction Hierarchy: LMS



# Abstraction

- Executive, Lead, and Manager are a type of Employee
- Thus they have a IS-A relationship with Employee
- Characteristics common to Executive/ Lead/ Manager are mentioned in Employee
- Example: Name, ID, DoB are common to all Employees of an organization (Executive, Lead, Manager)
- Lower hierarchy abstractions can have their own set of characteristics
  - Reporting\_Lead for Executive abstraction
  - Reporting\_Manager for Lead abstraction
  - List of Reporting Leads for Manager abstraction

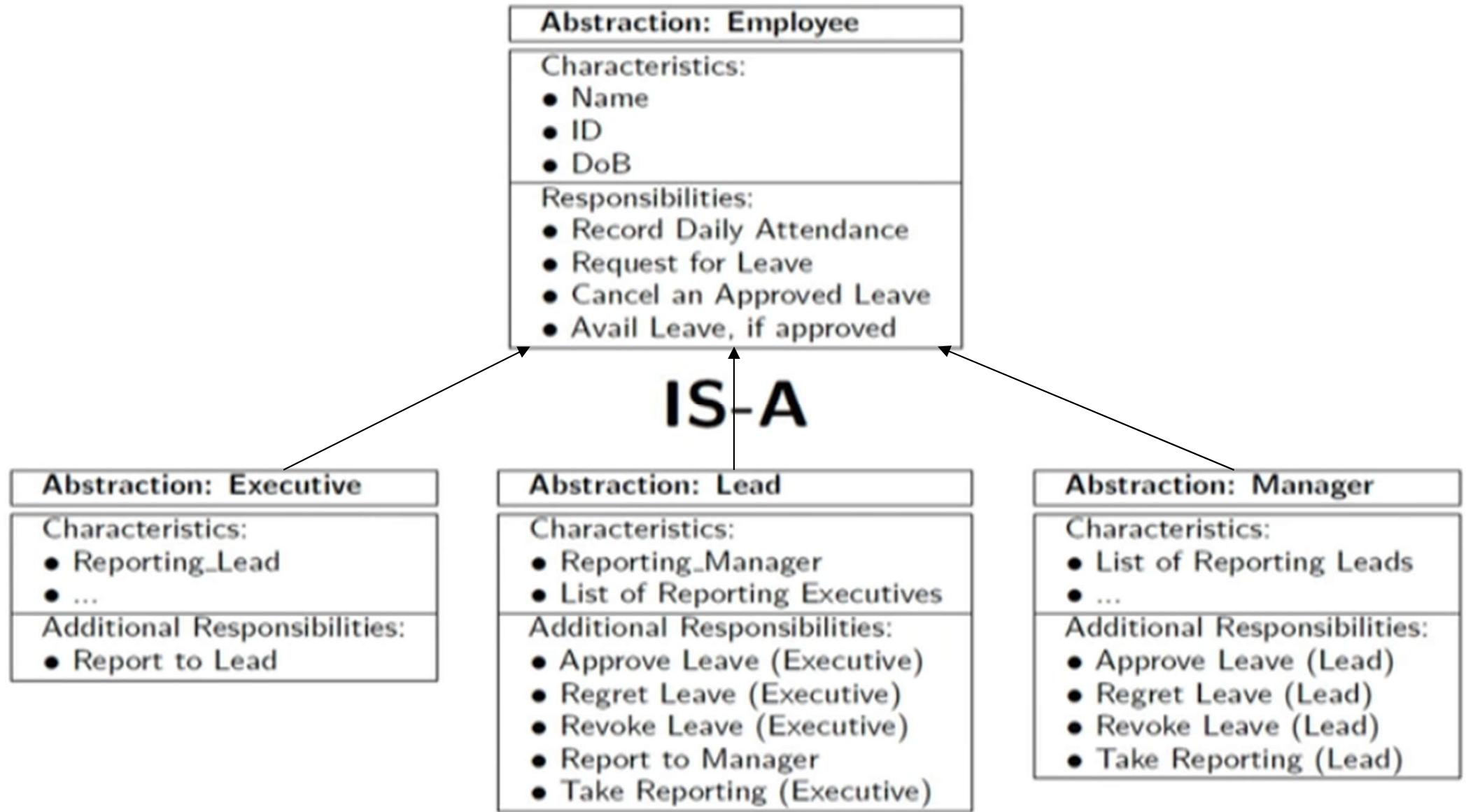
# Abstraction

- All lower level abstraction in the hierarchical structure are known as the Specialization
- Example: Executive is a specialization of Employee
- Whereas, Employee is the generalization of Executive, Lead, and Manager
- A generalized abstraction contains all the common characteristics of its specialization
- Specialization in turn can be certain specific characteristics which is not present in the generalized abstraction.

# Abstraction

- Example:
  - Generalized “Employee” have 4 responsibilities which are common for Executive, Lead, and Manager
  - Specialization “Executive” have 1 responsibility which is specific to the abstraction
  - Specialization “Lead” have 5 responsibilities which are specific to the abstraction
  - Specialization “Manager” have 4 responsibilities which are specific to the abstraction

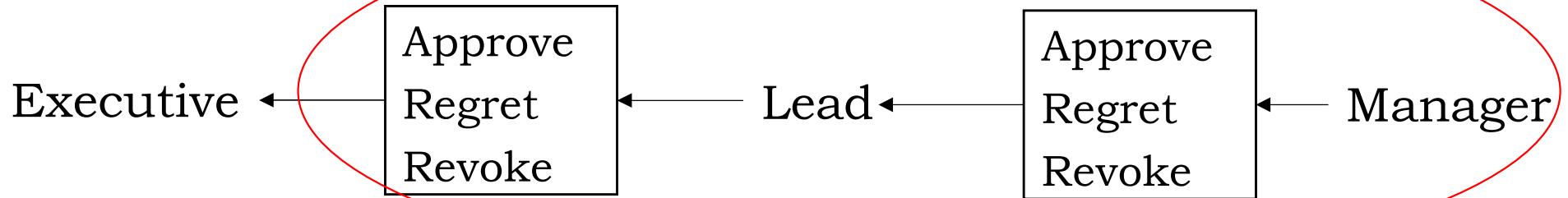
# Abstraction Hierarchy: LMS



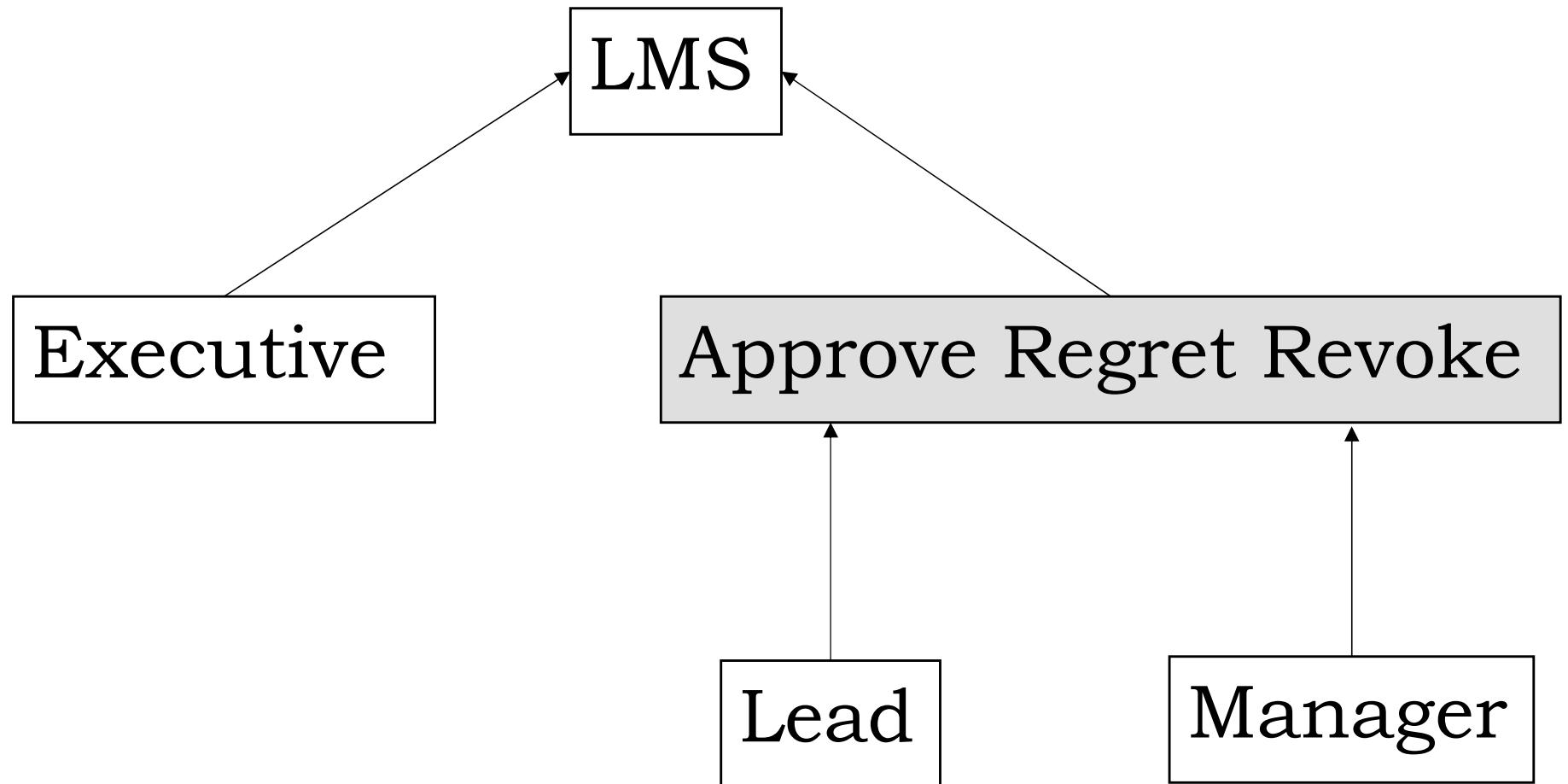
# Abstraction

- More commonalities can be observed
  - “Lead” **approve Leave** request from “Executive”
  - “Lead” **regret Leave** request from “Executive”
  - “Lead” **revoke Leave** request from “Executive”
  - “Manager” **approve Leave** request from “Lead”
  - “Manager” **regret Leave** request from “Lead”
  - “Manager” **revoke Leave** request from “Lead”

Commonalities ??



# Abstraction



Addition of a new concept in terms of a new abstraction hierarchy.  
Abstractions are better understood when put into hierarchy.

# Multi Level Hierarchy

## Abstraction: Sensor

### Characteristics:

- Location

### Responsibilities:

- Calibrate

IS-A

## Abstraction: Temperature Sensor

### Characteristics:

- Location
- Temperature

### Responsibilities:

- Calibrate
- Report current temperature

IS-A

## Abstraction: Active Temperature Sensor

### Characteristics:

- Location obtained from Sensor
- Temperature obtained from Temperature Sensor
- Setpoint

### Responsibilities:

- Calibrate
- Report current temperature
- Establish setpoint

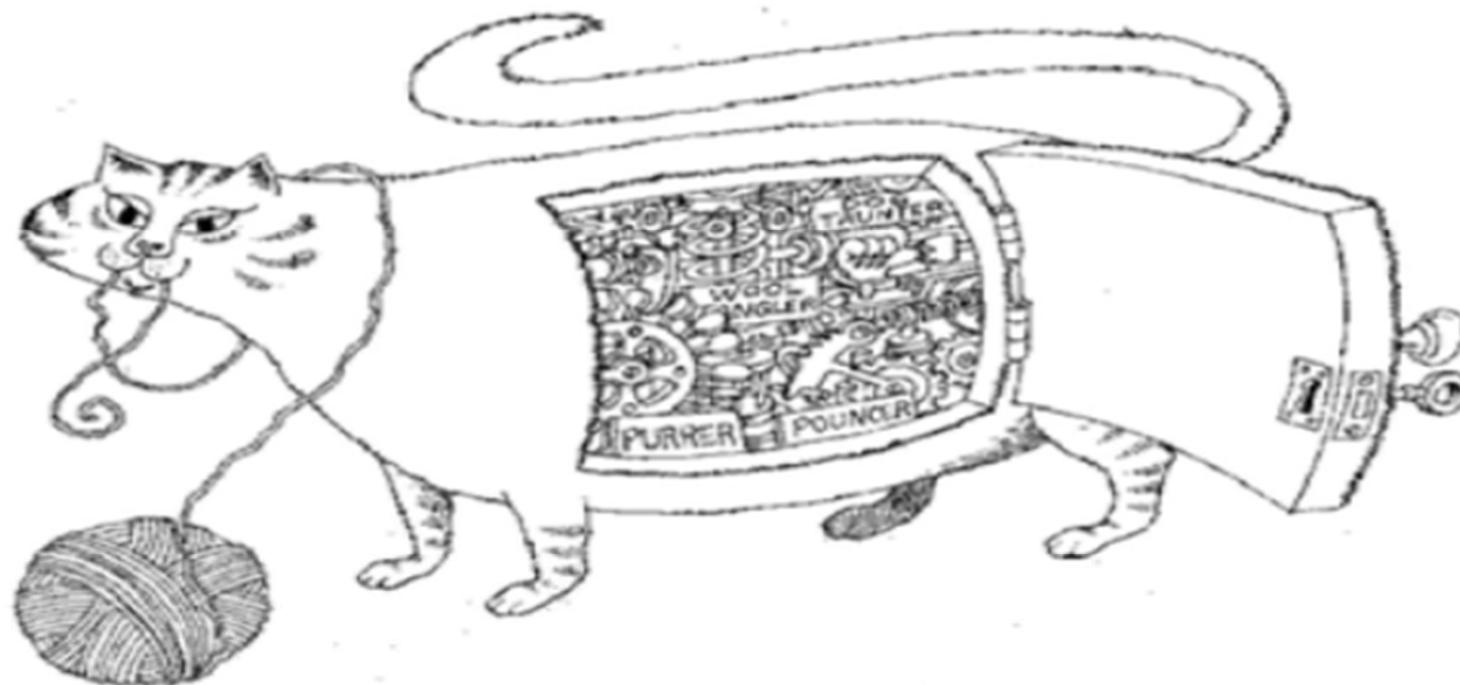
# Elements of Object Models

- Four major elements of the Object Model
  - *Abstraction*
  - **Encapsulation**
  - Modularity
  - Hierarchy
- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

# Encapsulation

- Primary task is to encapsulate
- The task is to put everything into one envelope
- By the term envelope we mean putting many different things together into a single box/module
- This strategy helps other developers to use it through a well planned approach
- It helps the elements of an abstraction to be grouped on basis of two aspects
  - Structure / characteristics / properties / variables
  - Behavior / responsibilities / tasks / methods

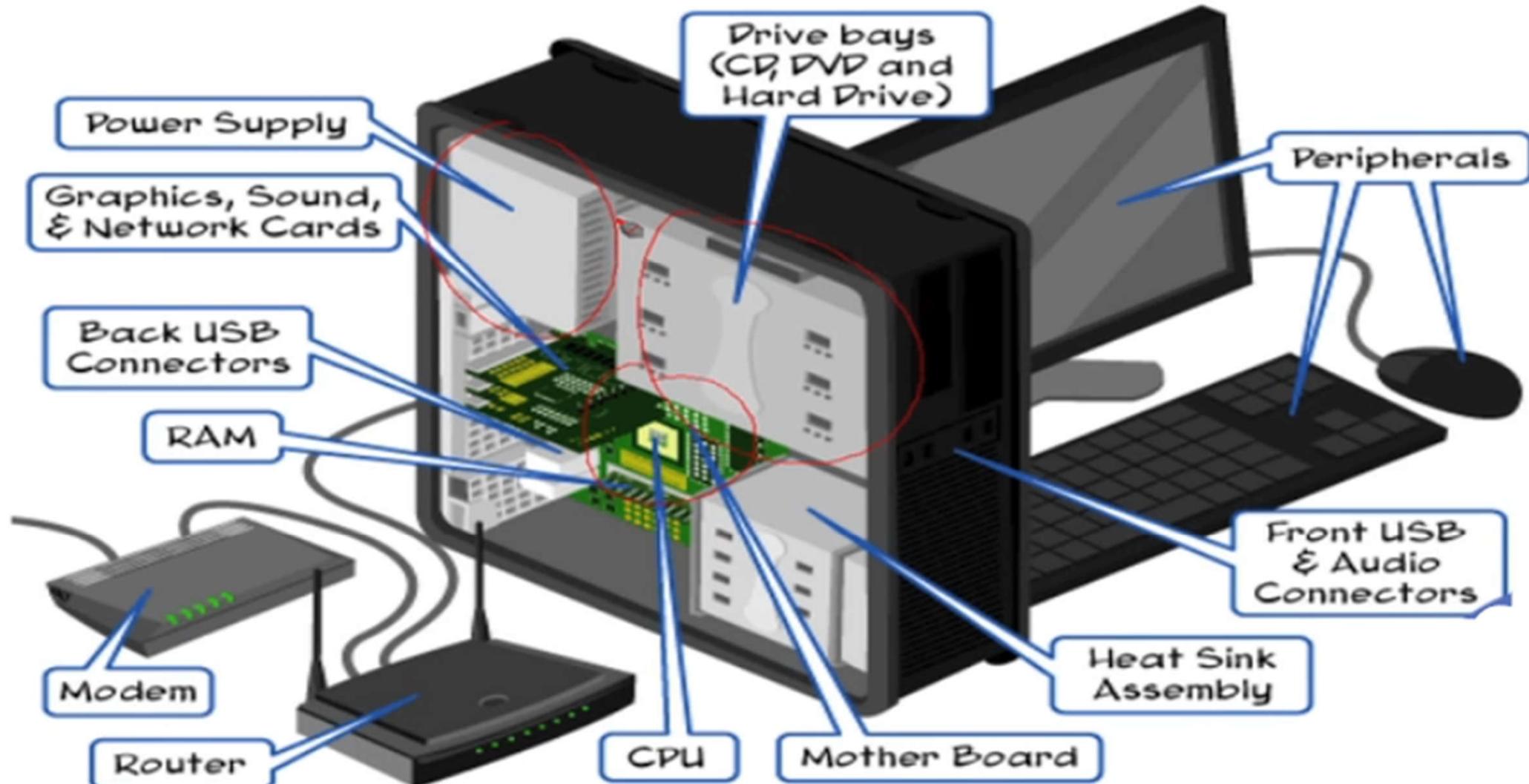
# Encapsulation: Example



*Encapsulation hides the details of the implementation of an object*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

# Encapsulation: Example



# Encapsulation: Example

- Structure is encapsulated in the implementation
  - **Store** for the elements
    - static or dynamic array
    - Single linked list
    - Deque
  - **Marker** for the top element
    - Array index
    - List pointer
    - Encapsulated in Deque
  - Hidden from the external world
- Behavior is exposed through interface
  - Interface for a stack
    - Push
    - Pop
    - Top
    - Empty
  - Exposed to the external world

Abstraction: Stack	
Structure:	<ul style="list-style-type: none"><li>- store</li><li>- marker</li></ul>
Behavior:	<ul style="list-style-type: none"><li>+ Push</li><li>+ Pop</li><li>+ Top</li><li>+ Empty</li></ul>

# Encapsulation

- For an Abstraction, the Encapsulation concept separates
  - Contractual Interface / Behaviors (Push – Pop – Top - Empty)
  - Implementation (Array - List - Deque)
- Encapsulation binds together the data and functions that manipulate the data
- Encapsulation keeps both (data and function) safe from the outside interference and misuse
- Data Abstraction => Data Hiding
- “Encapsulation” concept helps to perform “Abstraction” while Implementing a software.

# Encapsulation

- Remember that Encapsulation or hiding of the complex concepts are important
- For example a user would not care to understand the mechanism about how an ATM dispenses cash
- However, **if everything is encapsulated then the system cannot be used**
- Therefore, developing an interface for the user to interact and understand the basic working principle is important
- Thus both “Contractual Interface” (abstract) and “Implementation” (implement) are important.

# Encapsulation

- So the concept is similar to Client-Server model where Interface is what the Client gets to see and Server is the implementation details which remains encapsulated
- Interface provided to the Client always remains consistent with the underlying implementation
- For example: Buttons used for push/pop, etc. operation remains same
- Whereas, the procedure or code used to perform the push/pop operation might change
- Such change can be done in the server end or the implementation end which is encapsulated or hidden from the user/ client.

# Why the name “Contractual”?

- “Contractual” in the sense that we would confirm a set of operations that the user can perform in our software
- So that user gets an assurance for the set of tasks that it can perform. Discrete Systems !!
- Of course we can change the interface, but the basic working or tasks which needs to be performed by the user should be not be changed much
- **We can always modify the server part- which is hidden from the user (how the system worked)**
- Thus we say that the interface is kind of “Contracted” to perform certain fixed set of tasks.

# Encapsulation: Example

## Abstraction: Employee

### Structure:

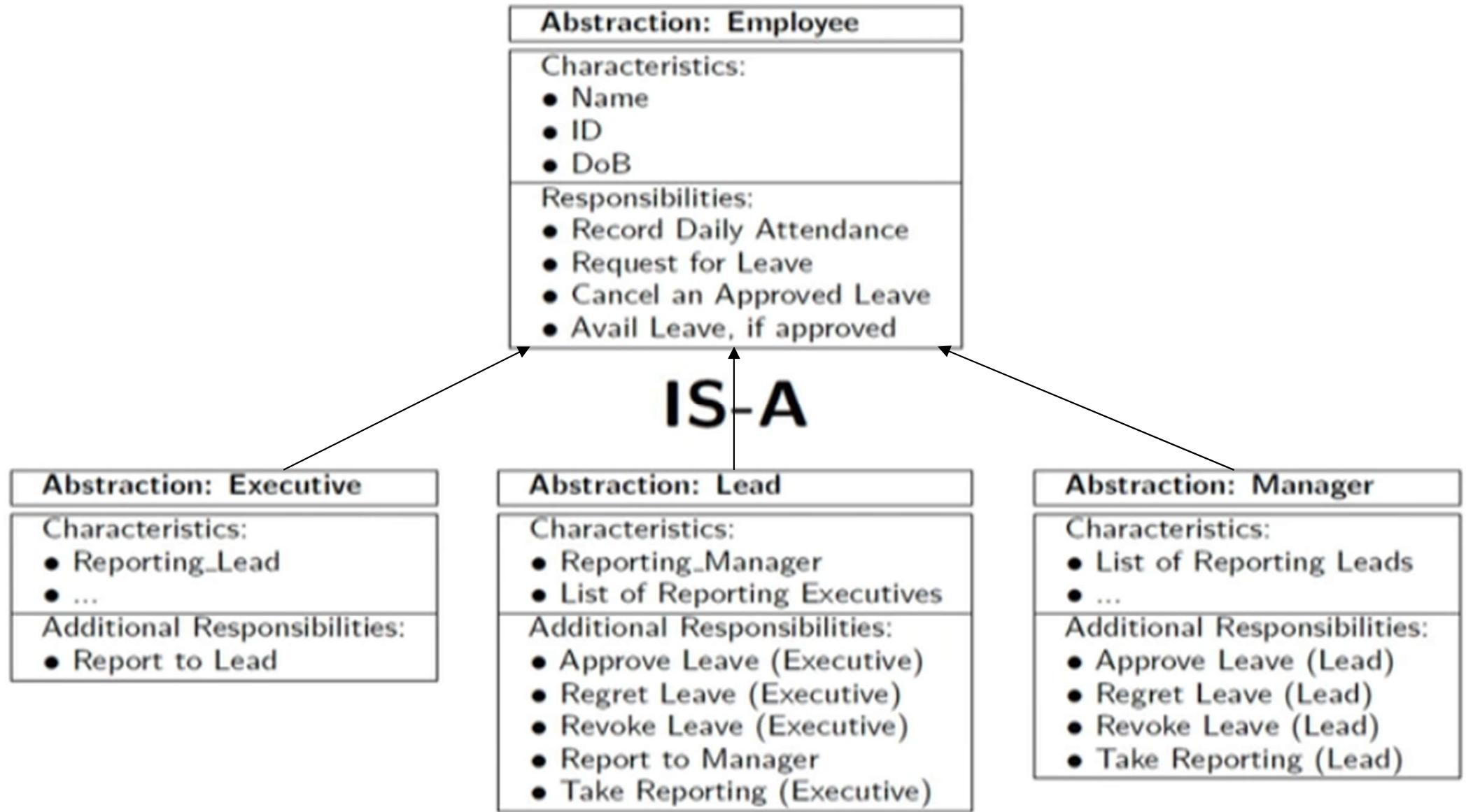
- Name
- ID
- DoB

### Behavior:

- + Record() // Daily Attendance
- + Request() // Leave
- + Cancel() // An Approved Leave
- + Avail() // Leave, if approved

- Structure = part of Implementation, and are Hidden
- Behavior = part of Interface, and are public anyone can use without understanding how “cancellation” is performed.

# Abstraction Hierarchy: LMS



# Abstraction vs Encapsulation

Abstraction	Encapsulation
Solves the problem in the design level	Solves the problem in the implementation level
Hide the unwanted data and expose useful data	Hide the code and data into a single unit to protect the data from outside world
Allows us to focus on what the object does instead of how it does it	Hide the internal details or mechanics of how an object does something
Provides outer layout, in terms of design	Provides inner layout used in terms of outer
Example: Look of a Mobile phone has a display screen and keypad buttons to dial a number.	Example: Implementation detail of a mobile phone deals with how keypad button and display screen connect with each other.

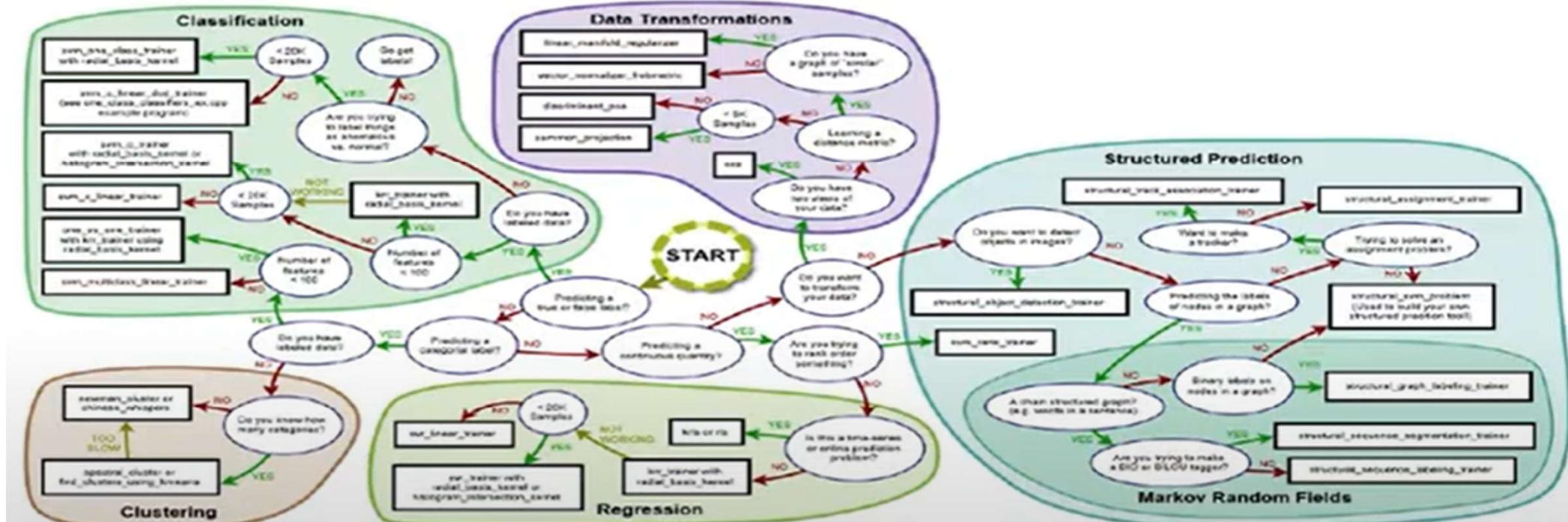
# Elements of Object Models

- Four major elements of the Object Model
  - *Abstraction*
  - *Encapsulation*
  - **Modularity**
  - Hierarchy
- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

# Modularity

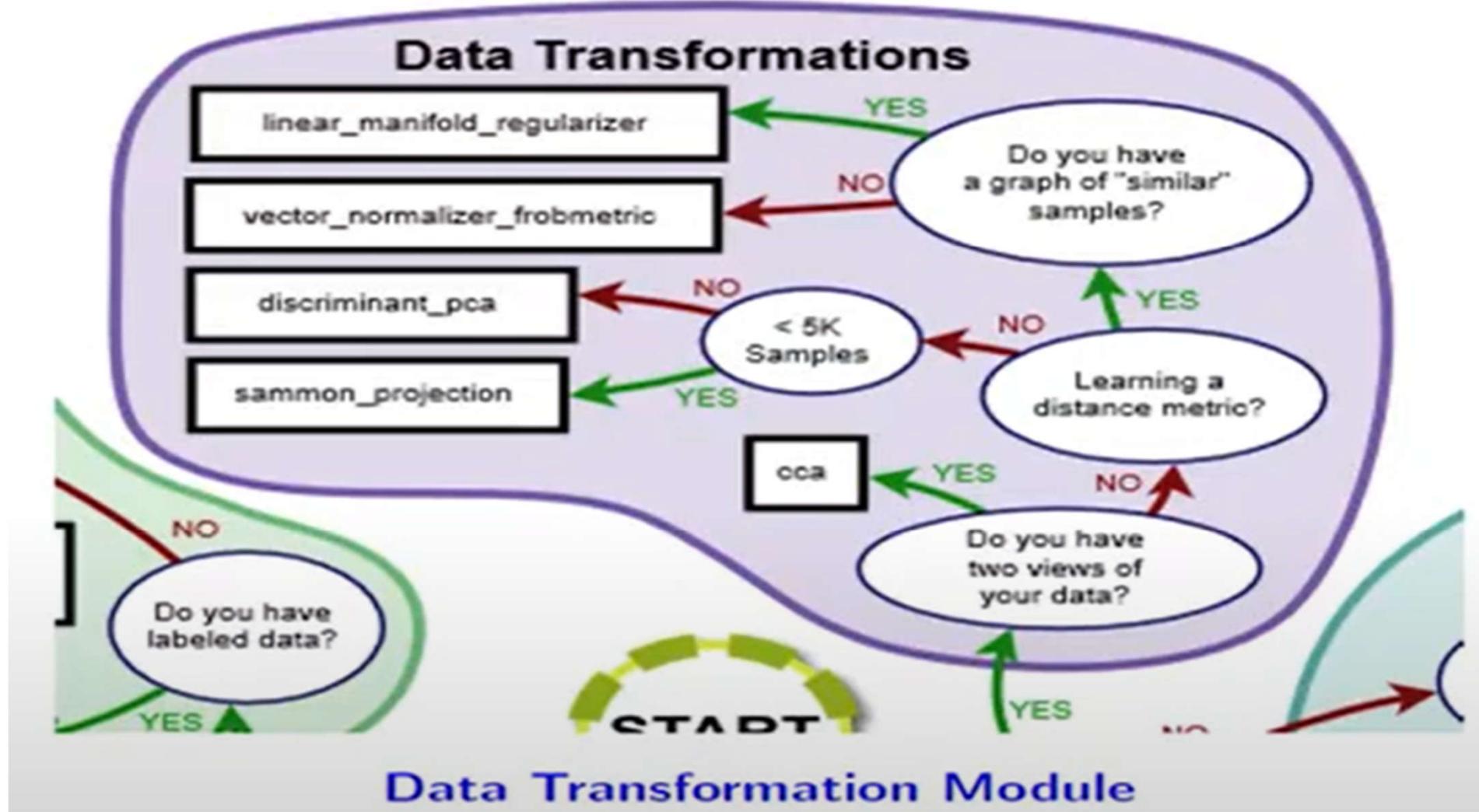
- Modularity depicts how the code and data are organized
- Modularity is to partition a program into individual components
- Modularity is done to manage complexity (reduce complexity)
- Module characteristics
  - Compile separately or together
  - Connected with the other modules

## Modularity: Example

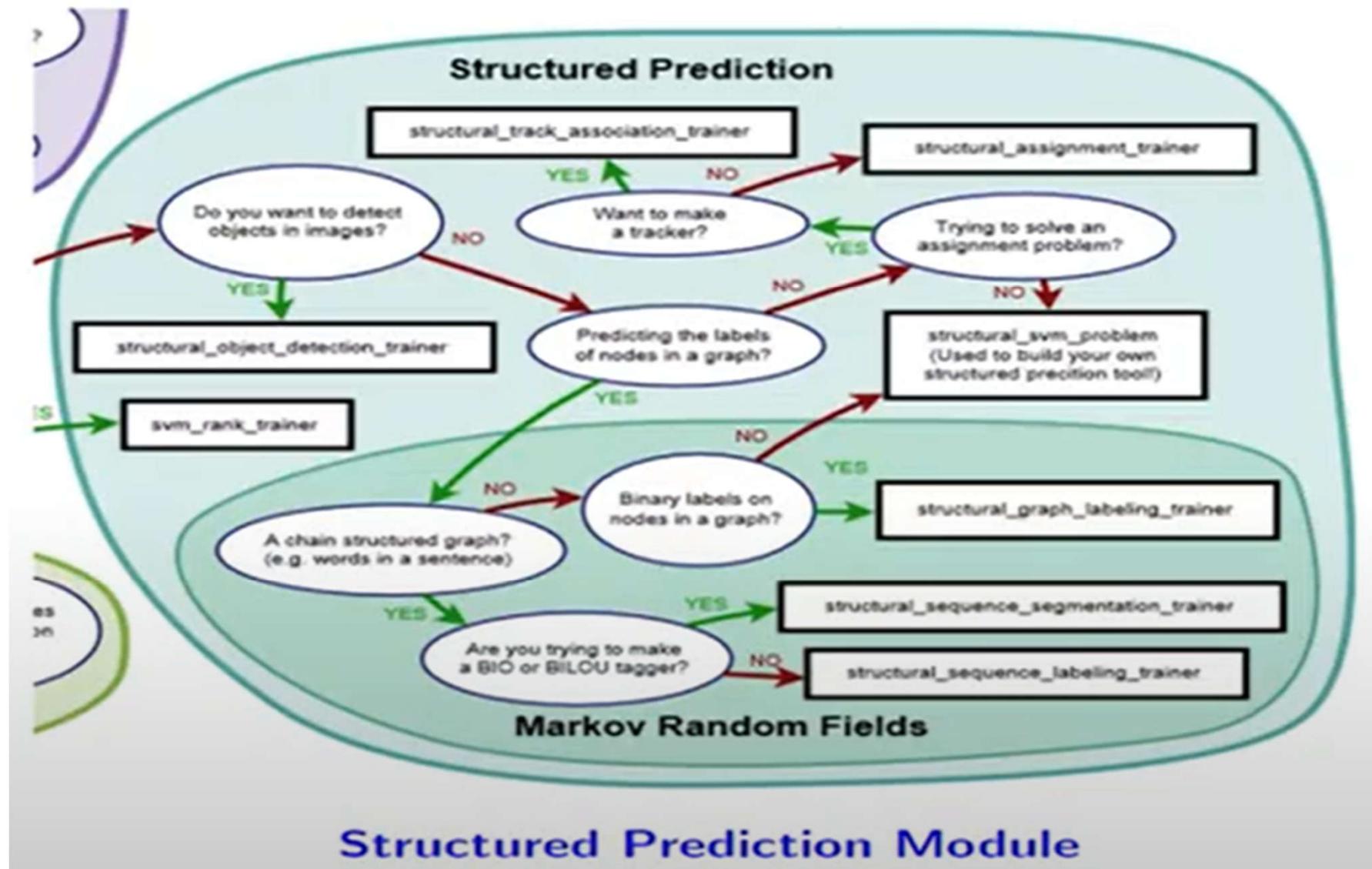


Modular decomposition of Machine Learning System

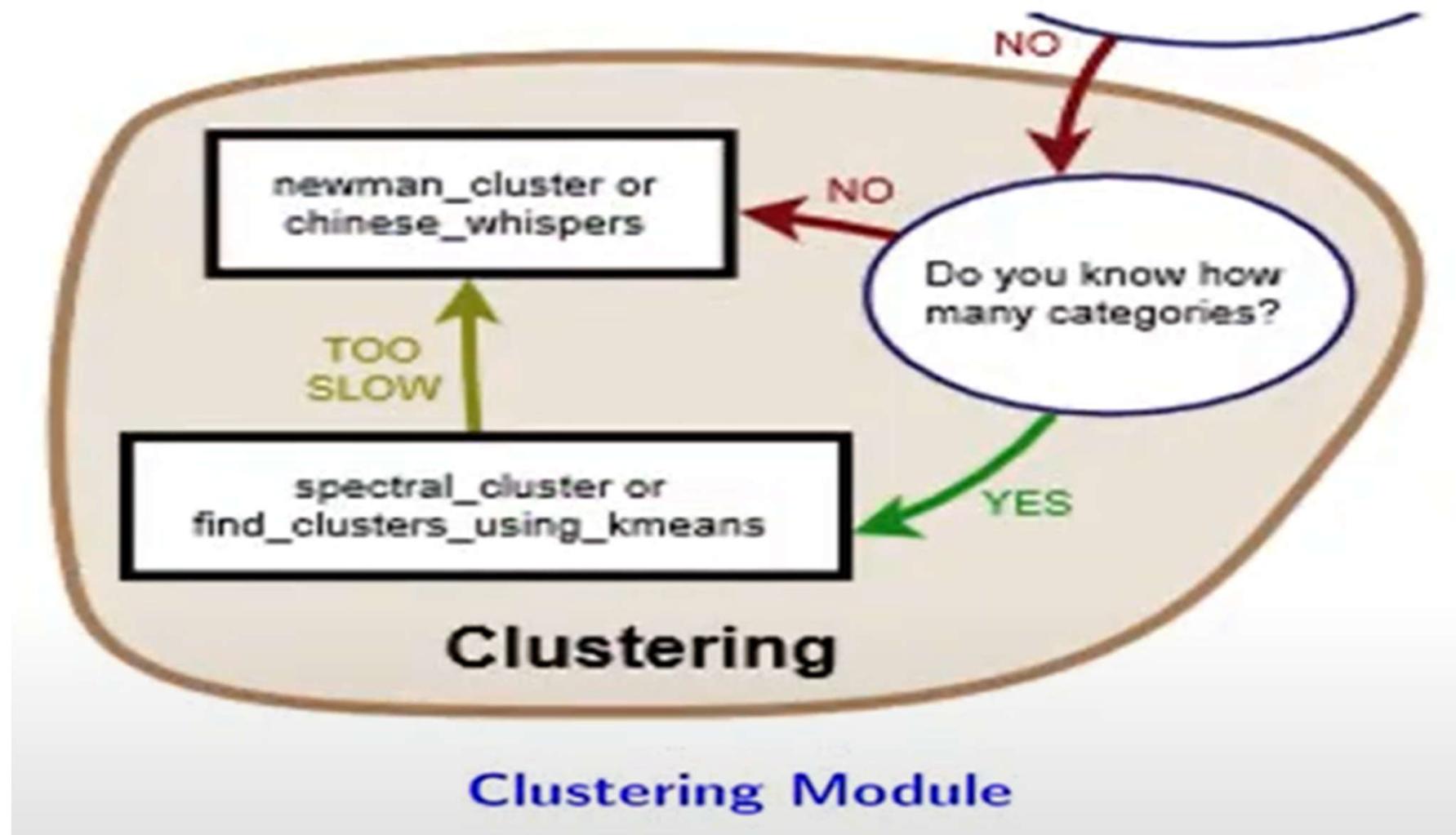
# Modularity: Example



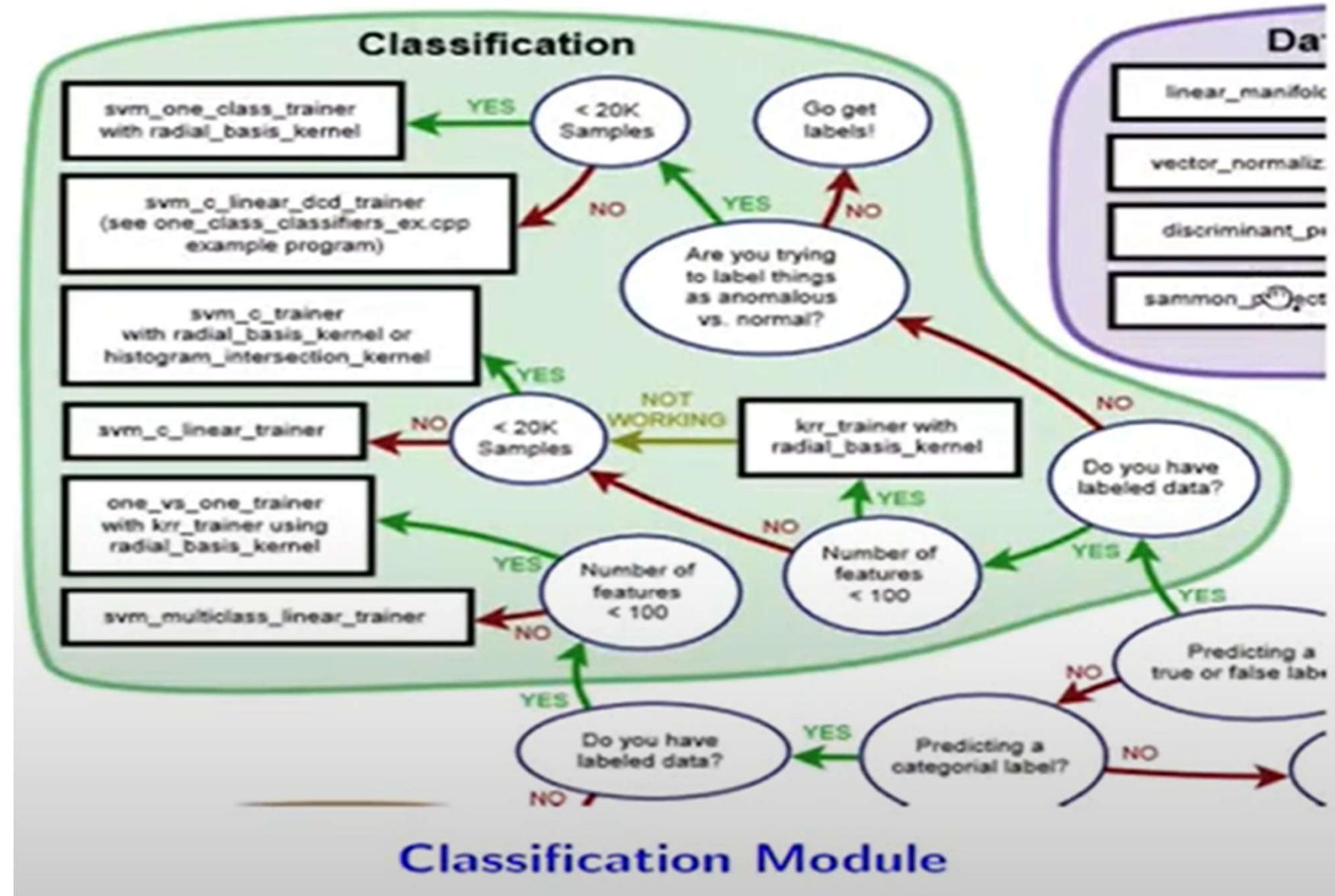
# Modularity: Example



# Modularity: Example



# Modularity: Example



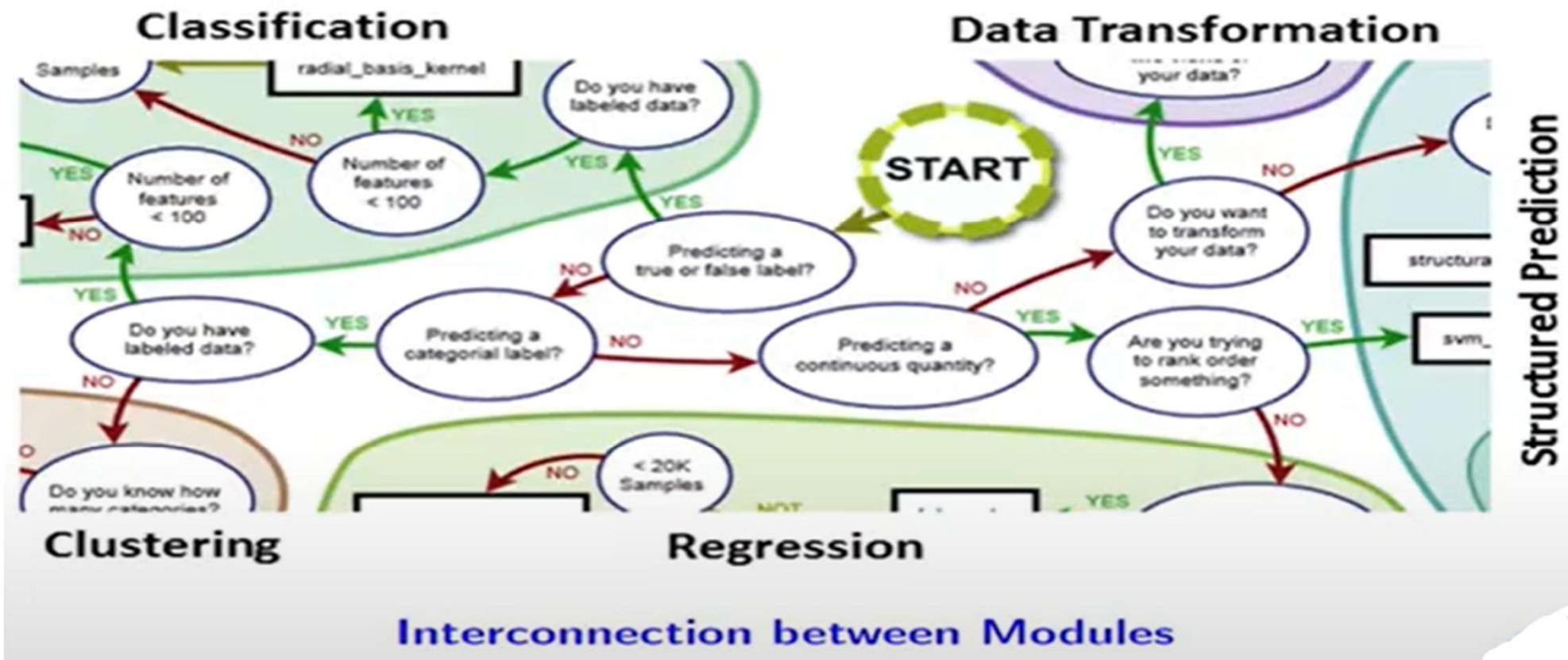
# Modularity: Example



**Regression**

**Regression Module**

# Modularity: Example



# Modularity: Example

- Modularity ensures
  - Partitioning the program codes into interrelated set of files and data
  - Also maintain the interconnections between the partitioned program codes
  - For example: the below sub-components performs the task of interconnection between the modules.



# Modularity

- It is always easier to look into the system when modularized
- It is easier to divide the tasks when modularization is assured
- There can be hierarchy in a modularized structure.

# Modularity: Objectives

- Modular Decomposition

- Systematic mechanism to divide problem into individual components
- We only put together a set of code into one module which have some sort of well-defined functionality
- So that together they represent a certain functionality/ concept

- Modular Composability

- Modularization should be done in such a way that we can reconstruct it back
- Combine smaller modules into a bigger module if required
- Enables reuse of existing components.

# Modularity: Objectives

- Modular Understandability

- Understand each module as a unit
- Example: For performing i/o operation lets include stdio.h !!
- Here we do not know what is the underlying structure of stdio.h, but what we know is a concept that for i/o related operations we need to include all functions within stdio.h .. This is understandability.

# Modularity: Objectives

## ■ Modular Continuity

- We need to make the modules in such a way that it is easier to make changes
- Example: Bug fixation, Change in specification, Change in Business, etc.
- So we should not end-up in a situation where for one change, we need to modify multiple module (i.e. change across modules)
- Change should be limited to one module or at least a smaller number of modules
- Thus it reduces side-effects.

# Modularity: Objectives

- Modular Protection
  - Errors should be localized
  - If there is an error “divide by zero” then that should be localized to math.h
  - If there is an error “memory allocation failure” then that should be localized to memory.h and so on
  - Errors should not spread across multiple modules.

# Modularity: Challenges

- Decomposition into smaller module may be difficult
- Arbitrary modularization is sometimes worse than no modularization
- Modularization should follow the **semantic grouping** of **common** and **interrelated functionality** of the system.

# Modularity: Example

- Consider a Distributed System
  - Runs on a distributed set of processors
  - Uses a message passing mechanism to coordinate the activities of different programs
  - Has several hundreds of event & few thousand kinds of messages
- What should be the strategy to modularize?
  - Naïve strategy might be to define each message class in own module
  - This strategy is poor because it would end up into numerous modules
  - Documenting the system would be a nightmare !!
  - Difficult to find classes
  - When decisions change, updating hundreds of modules are required
  - Thus hiding information becomes a problem.

# Intelligent Modularization

- Group logically related abstraction
- Minimize dependency among modules
- Simple enough to understand fully
- Ease of change
  - Possibility to change the implementation strategy of an individual module
  - Without enough knowledge of other modules
  - Without effecting the working of other modules
  - Change in a module for an already implemented system needs reasonable justification of the benefits for the change.

# Elements of Object Models

- Four major elements of the Object Model
  - Abstraction
  - Encapsulation
  - Modularity
  - **Hierarchy**
- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - Concurrency
  - Persistence

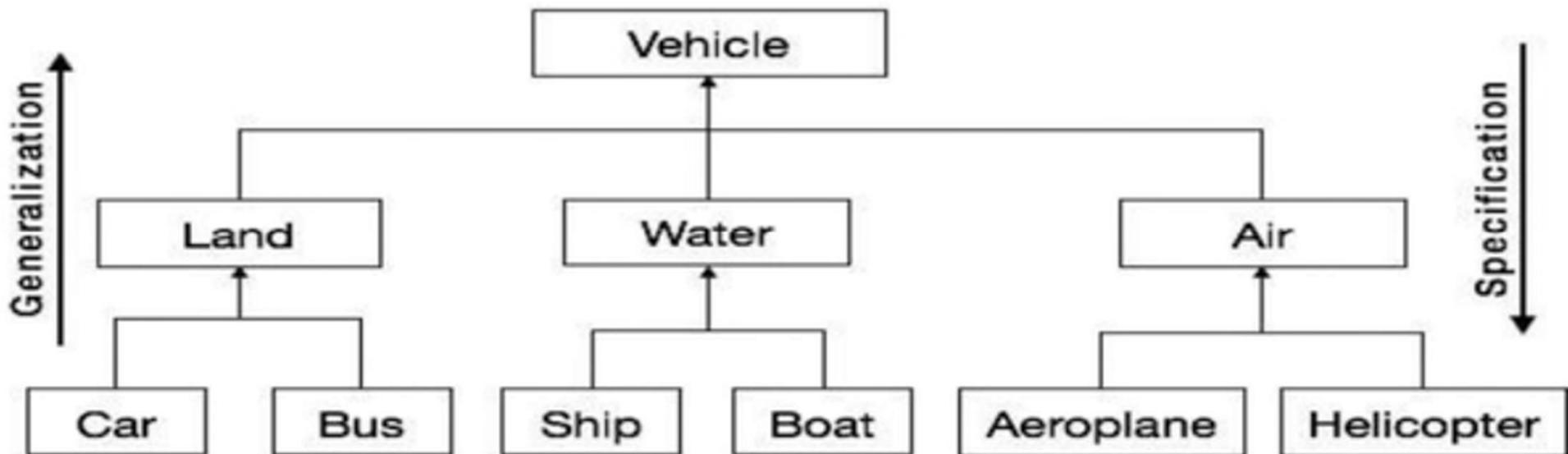
# Hierarchy

- Hierarchy is a ranking or ordering of abstractions
- The set of abstraction that a system have moves from “less details” to “more details”
- The bottom-most level have highest level of details, and top-most level have the minimal details



# Hierarchy

- Two most important hierarchies in a complex system are
  - Class structure or **is-a** hierarchy (Abstraction or IS-A)
  - Object structure or **part-of** hierarchy (Decomposition or **HAS-A**)
- Common structure and behavior are migrated to superclass
  - Superclass represent generalized abstraction
  - Subclass represent specializations
  - Subclass can add, modify and hide methods from superclass

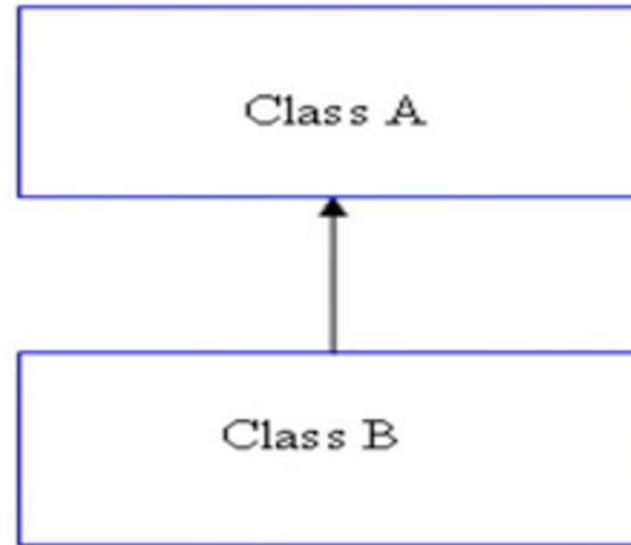


# Hierarchy

- Hierarchy provides an opaque barrier to hide the methods and state
- Inheritance is required to open this interface and allow state and methods to be accessed between two levels of abstraction
- Different programming languages trade-off support for inheritance in different ways
- C++ and Java offer great flexibility
  - Private variables/ modules – accessible only to the class
  - Protected variables/ modules – accessible only to the class and its subclasses
  - Public variables/ modules – accessible to all clients.

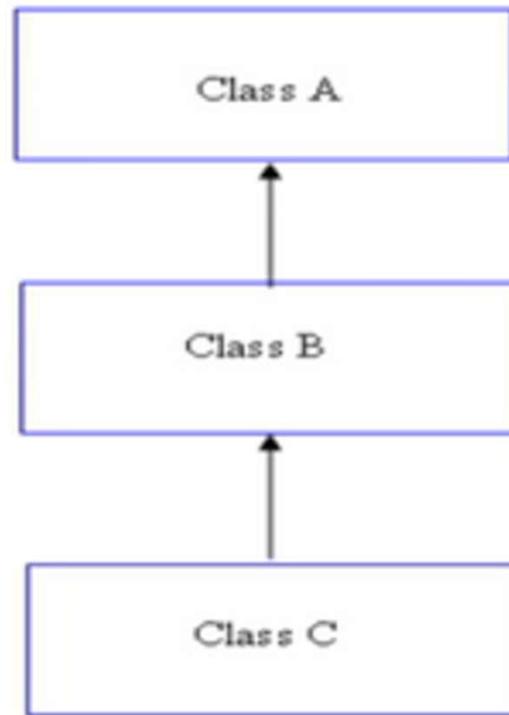
# Single Inheritance

- A single superclass is inherited by a single subclass



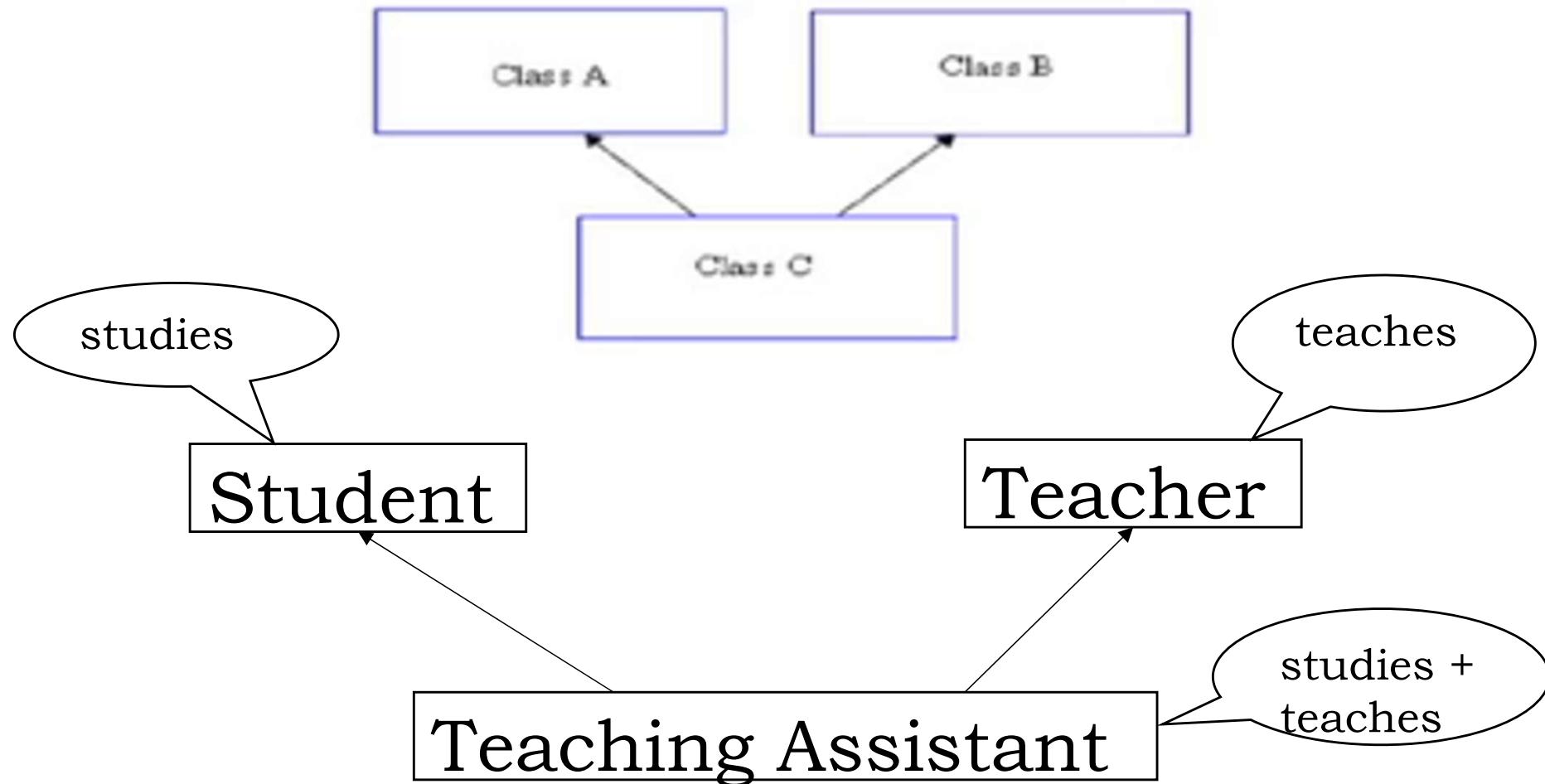
- Class B **IS-A** Class A
- Class B is the specialization of Class A
- Class A is the generalization of Class B

# Multi-level Inheritance



# Multiple Inheritance

- For certain abstractions, it is important to provide inheritance from multiple superclasses

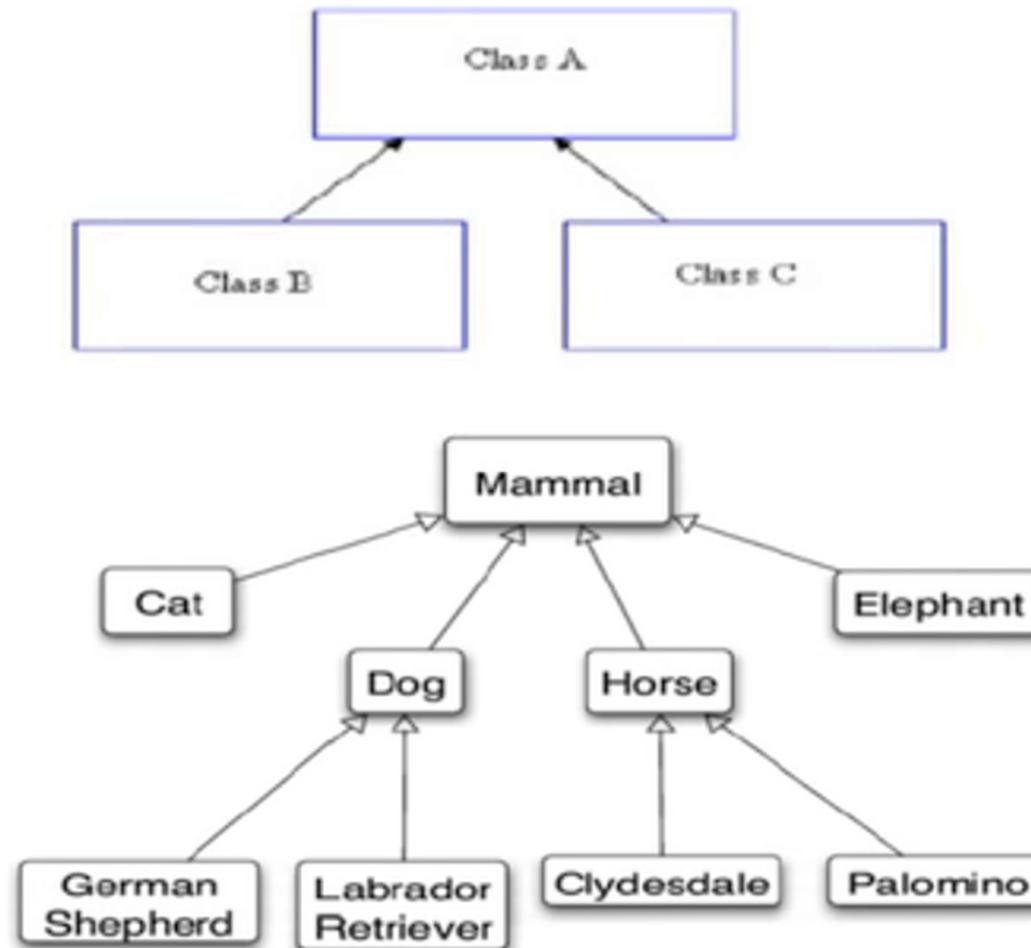


# Multiple Inheritance: Challenges

- A very critical and important concept related to Hierarchy
- It is easy to identify the multiple inheritances (or where we need multiple inheritances)
- But there are a lot of fundamental, theoretical and conceptual factors which one need to look into while dealing with multiple inheritance
- For example if we inherit the same method from two different superclasses, then what would be the behavior of the method in the sub-class.

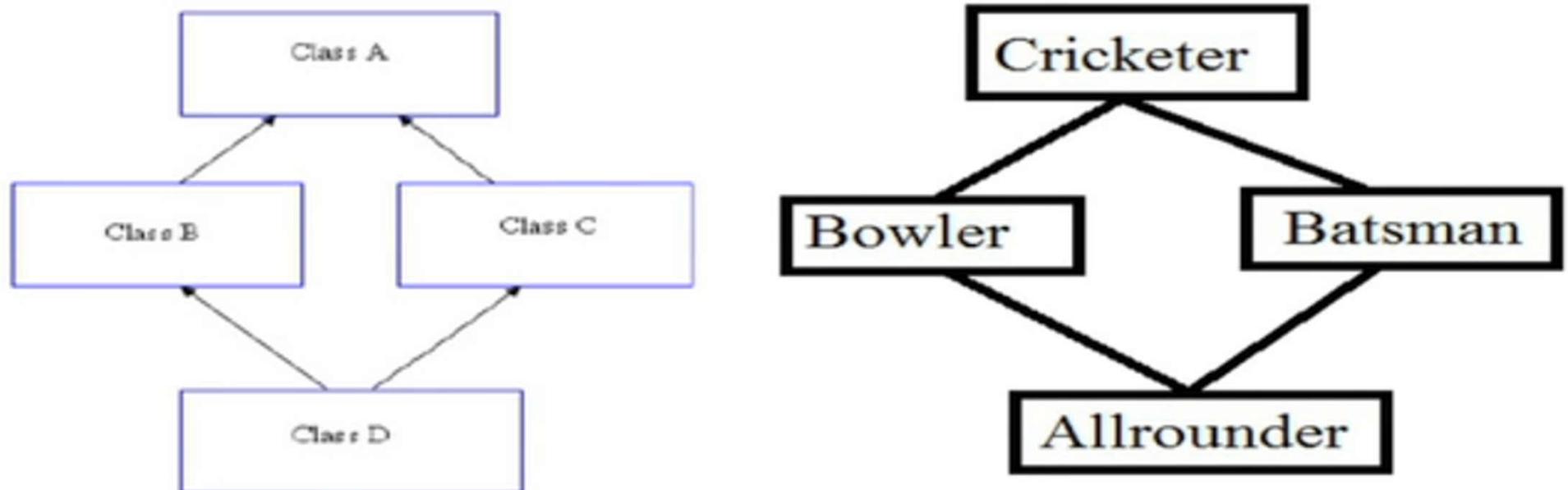
# Hierarchical Inheritance

- More than one sub classes are created from the same super class



# Hybrid Inheritance

- It may be a combination of
  - Multilevel and Multiple inheritance
  - Hierarchical and Multilevel inheritance
  - Hierarchical and Multiple inheritance



# Single Inheritance - Example

```
class one
{
    int x;
    public void print_1()
    {
        x = 10;
        System.out.println("CSE");
    }
}

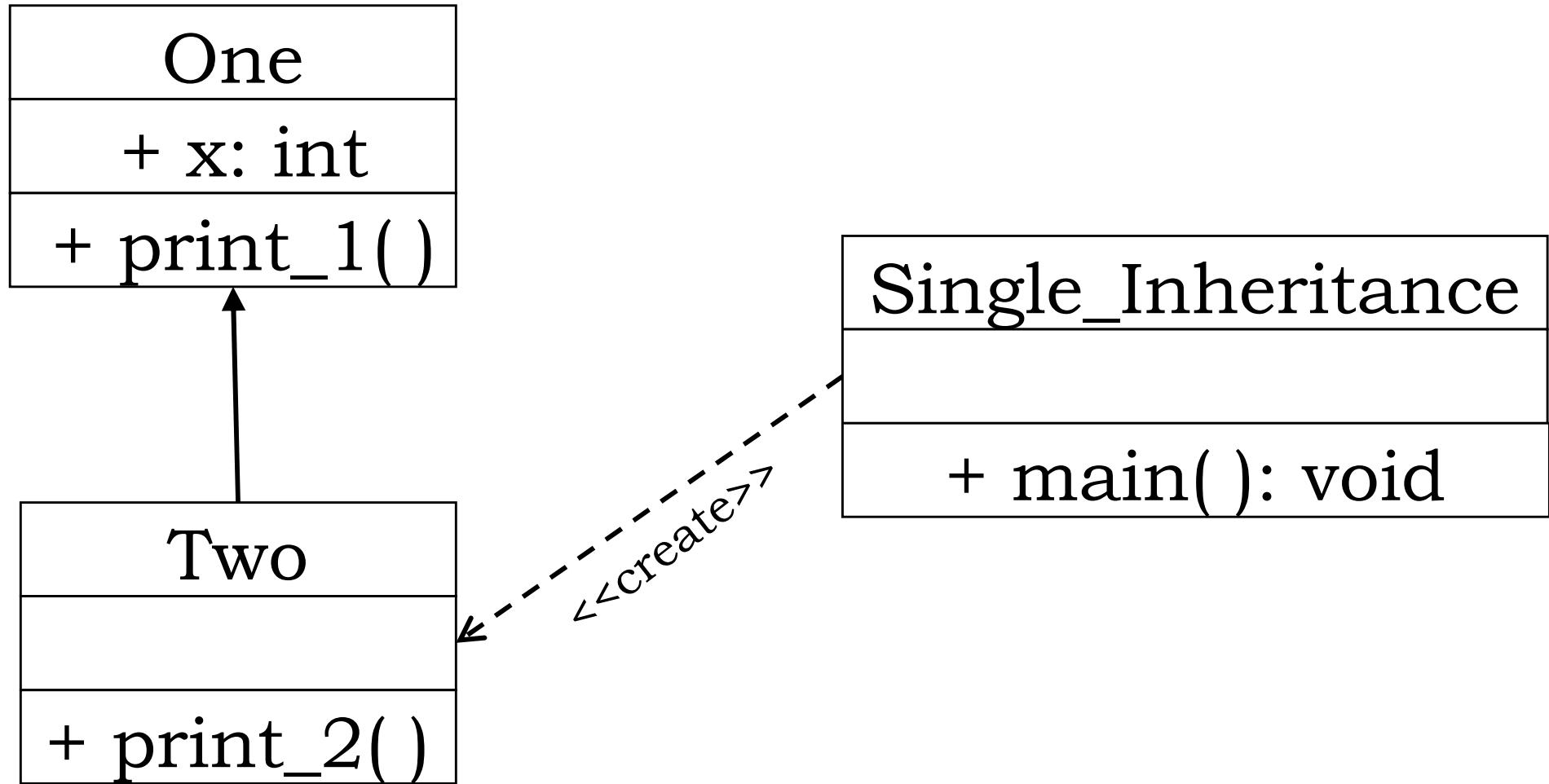
class two extends one
{
    public void print_2()
    {
        System.out.println("IIIT Vadodara");
    }
}
```

```
public class Single_Inheritance
{
    public static void main(String[] args)
    {
        two g = new two();
        g.print_1();
        g.print_2();
        System.out.print(g.x);
    }
}
```

## OUTPUT:

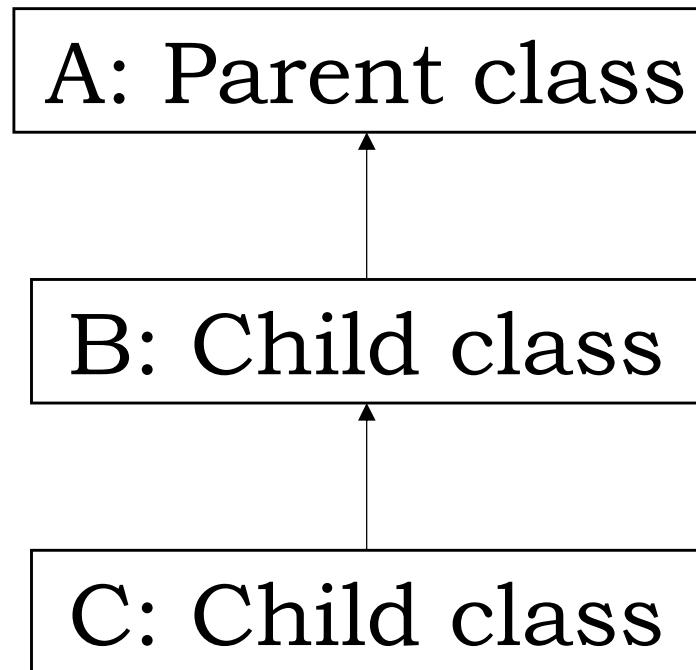
```
CSE
IIIT Vadodara
10
```

# Single Inheritance – Class Diagram



# Multilevel Inheritance

- A child class derives member variables and methods from another derived class.



# Multilevel Inheritance - Example

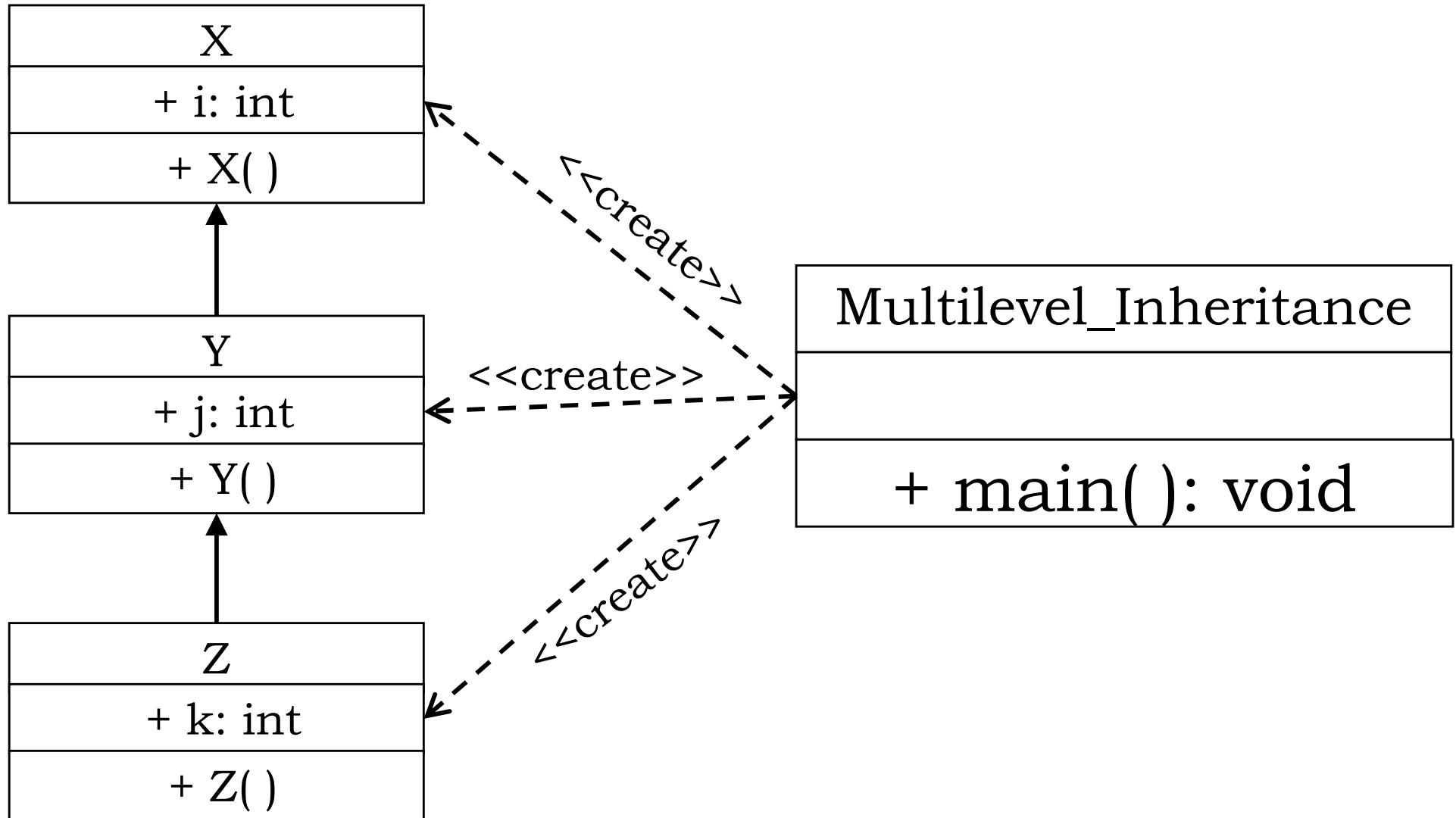
```
class X
{
    int i;
    X( )
    {
        i=5;
    }
}

class Y extends X
{
    int j;
    Y( )
    {
        j=7; i++;
    }
}
```

```
class Z extends Y {
    int k;
    Z( )  {
        k=9; j++;
    }
}

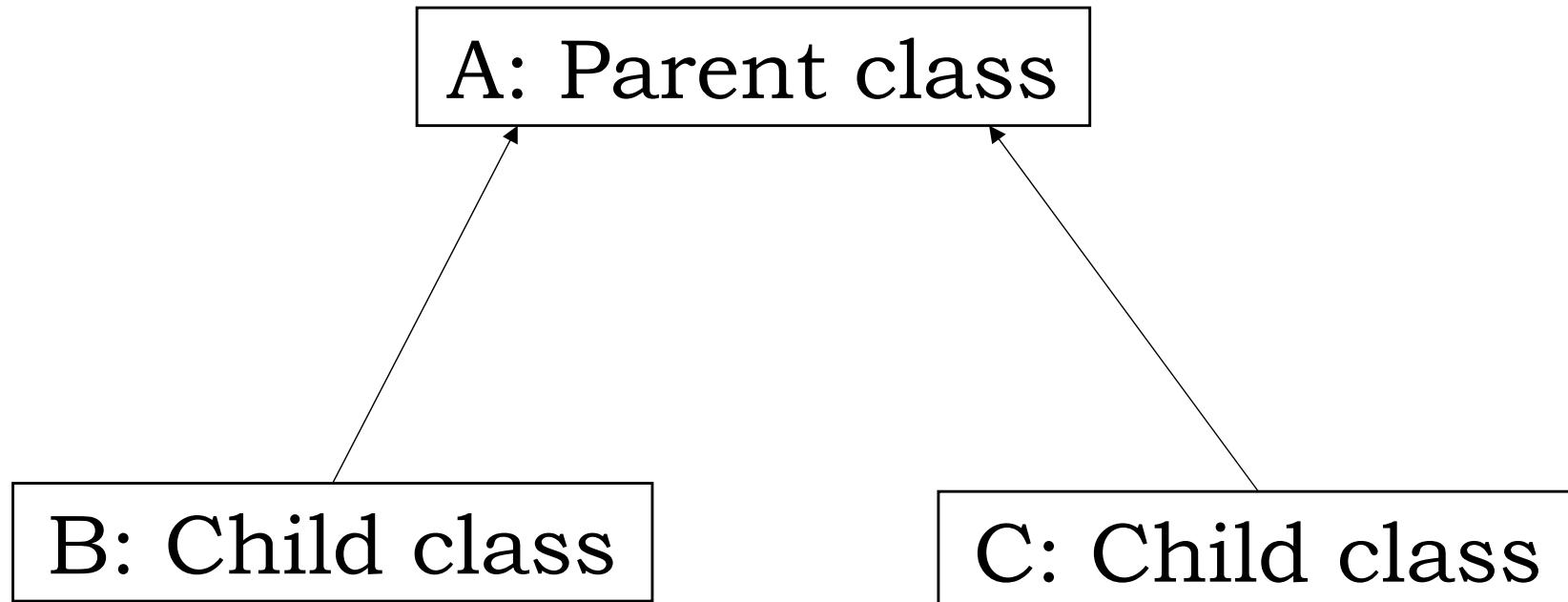
public class Multilevel_Inheritance{
    public static void main(String args[]) {
        X a = new X();
        Y b = new Y();
        Z c = new Z();
        System.out.println(a.i);
        System.out.println(b.i + b.j);
        System.out.println(c.i + c.k);
        System.out.println(c.i + c.j + c.k);
    }
}
```

# Multilevel Inheritance – Class Diagram



# Hierarchical Inheritance

- One base class is derived by many child classes.



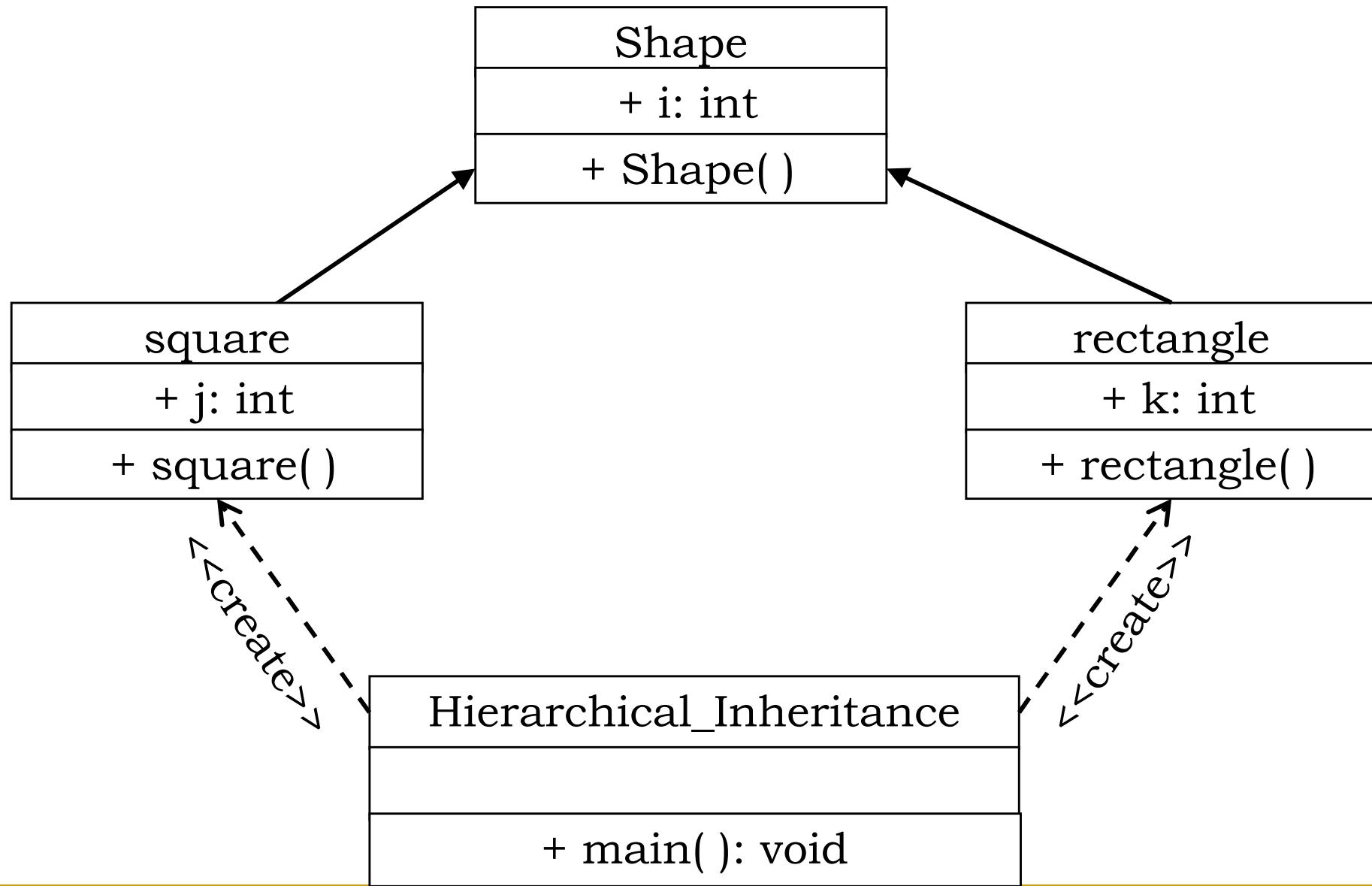
# Hierarchical Inheritance - Example

```
class Shape  
{  
    int i;  
    Shape( )  
    {  
        i=5;  
    }  
}
```

```
class square extends Shape  
{  
    int j;  
    square( )  
    {  
        j=6;  
    } }
```

```
class rectangle extends Shape {  
    int k;  
    rectangle( )  
    {  
        k=7;  
    }  
}  
  
class Hierarchical_Inheritance {  
    public static void main(String args[]) {  
        Shape a = new Shape();  
        square b = new square();  
        rectangle c = new rectangle();  
        System.out.println(a.i);  
        System.out.println(b.i * b.j);  
        System.out.println(c.i + c.k);  
    } }
```

# Hierarchical Inheritance – Class Diagram



# Constructors

```
class Shape  
{  
    int i;  
    Shape( )  
    {  
        i=5;  
    }  
}
```

```
class square extends Shape  
{  
    int j;  
    square( )  
    {  
        j=6;  
    } }
```

```
class rectangle extends Shape {  
    int k;  
    rectangle( )  
    {  
        k=7;  
    }  
}  
  
class Hierarchical_Inheritance {  
    public static void main(String args[]) {  
        Shape a = new Shape();  
        square b = new square();  
        rectangle c = new rectangle();  
        System.out.println(a.i);  
        System.out.println(b.i * b.j);  
        System.out.println(c.i + c.k);  
    } }
```

# Constructors

- It is a method which gets initialized at the time of object creation
- *New* keyword used to instantiate an object also initializes the constructors
- Do not need objects to call them
- They have same name as that of the containing Class
- Multiple constructors may exist in a single Class
- Multiple constructors should have different parameter list.

# Constructors

- They do not have a return type
- They returns instance of the class i.e. the new object
- If a class is not having a constructor,
- Java creates a *default constructor*
- *Default constructor* with no parameters sets all fields to zero
- Any variable declared within a constructor is Local to the constructor
- No other constructor can access a variable local to another constructor.

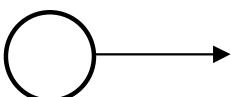
# Constructor example

```
public class Point {  
    int x;  
    int y;  
  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
    ...  
}
```

# Tracing a constructor call

- What happens when the following call is made?

```
Point p1 = new Point(7, 2);  
p1.translate(5, 10);
```

*p1* 

x 

y 

```
public Point(int initialX, int initialY) {  
    x = initialX;  
    y = initialY;  
}  
  
public void translate(int dx, int dy) {  
    x = x + dx;  
    y = y + dy;  
}
```

# Common constructor bugs

1. Re-declaring fields as local variables ("shadowing"):

```
public Point(int initialX, int initialY) {  
    int x = initialX;  
    int y = initialY;  
}
```

- ❑ This declares local variables with the same name as the fields, rather than storing values into the existing variables. The existing attributes remain 0.

2. **Accidentally** giving the constructor a return type:

```
public void Point(int initialX, int initialY) {  
    x = initialX;  
    y = initialY;  
}
```

- ❑ This is actually not a constructor, but a method named Point

# Constructor example

```
public class Point {  
    int x;  
    int y;
```

```
public Point(int initialX, int initialY) {  
    x = initialX;  
    y = initialY;  
}
```

```
public void translate(int dx, int dy) {
```

```
    x = x + dx;  
    y = y + dy;  
}
```

...

```
}
```

# Constructor example

```
public class PointMain3 {  
    public static void main(String[] args) {  
        // create two Point objects  
        Point p1 = new Point(5, 2);  
        Point p2 = new Point(4, 3);  
  
        // print each point  
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");  
  
        // move p2 and then print it again  
        p2.translate(2, 4);  
  
        // print each point once again  
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");  
    }  
}
```

# Multiple constructors

- A class can have multiple constructors.
  - Each one must accept a unique set of parameters.
- *Exercise:* Write a Point constructor with no parameters that initializes the point to (0, 0).

```
// Constructs a new point at (0, 0).
public Point() {
    x = 0;
    y = 0;
}
```

# Constructor example

```
public class Point {  
    int x;  int y;  
  
    public Point() {  
        x = 0;  
        y = 0;  
    }  
    public Point(int initialX, int initialY) {  
        x = initialX;  
        y = initialY;  
    }  
  
    public void translate(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
}
```

# Constructor example

```
public class PointMain3 {  
    public static void main(String[] args) {  
        Point p1 = new Point(5, 2);  
        Point p2 = new Point(4, 3);  
        Point p3 = new Point();  
  
        // print each point  
        System.out.println("p1: (" + p1.x + ", " + p1.y + ")");  
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");  
        System.out.println("p3: (" + p3.x + ", " + p3.y + ")");  
  
        p2.translate(2, 4);  
        System.out.println("p2: (" + p2.x + ", " + p2.y + ")");  
    }  
}
```

# Types of Constructors

- Parameterized Constructor
- Non parameterized constructors
- Default constructors
- **Copy constructors**
- Constructor overloading

# Copy Constructor

- A copy constructor is a method/constructor that initialize an object using another object within the same class
- Enables to get a copy of an existing object
- Copy constructors take only one parameter
- The parameter is a reference of the same class.

# Copy Constructor

- Advantages of using copy constructors:
  - Copying an object with multiple attributes
  - when you have a complex object with many attributes it is much simpler to use the copy constructor
  - if you add an attribute to your class, you just change the copy constructor to take this new attribute into account instead of changing every occurrence of the other constructor.
- Copying the constructors can be performed by:
  - Shallow copy
  - Deep copy

# Copy Constructor

## ■ Shallow copy

- Creates a copy of an object by copying data of all member variables as it is
- `Point p1 = new Point(5,9);`
- `Point p2 = p1; //copy`
- Objects p1 and p2 are referencing to the same memory location
- Changes made in one object will effect the other
- What if we want to Copy but create a separate entity?

# Copy Constructor

- Deep copy
  - Dynamically allocates memory for the new object
  - Copies existing entities of the existing object
  - Entities of the existing object is duplicated to the new object
  - Both objects are located at different memory locations
  - There is no connection between the two object
  - Subsequent changes made to one does not affect the other
  - Also known as user-defined copy constructor.

# Elements of Object Models

- Four major elements of the Object Model
  - *Abstraction*
  - *Encapsulation*
  - Modularity
  - Hierarchy
- Three minor elements of the Object Model (useful but not essential)
  - **Typing**
  - Concurrency
  - Persistence

# Minor Elements

- These are not essential elements
- Depending upon the skill of the designer
- Depending upon the Specification of the system to be developed
- Such minor elements may be considered.

# Typing

- This is NOT the Word/Code typing speed !!
- Typing is the representation of the type of a class or type of an object associated with a class
- It is important because
  - Usually in programming we have variables
  - Variable have a definite type or datatype
  - We know what type of data we need to store and accordingly define the types
  - However, for an object the scenario is different
  - It is not going to have a single value
  - It is going to represent a class
  - Inherent properties of the class which the object refers to needs to be understood
  - Inherent properties of the class which is being referred by an object is the Type of the Object.
- Typing is the enforcement of the class of an object such that objects of different types
  - May not be interchanged
  - May be interchanged only in very restricted ways

# Typing

- Type derives from the theory of abstract data types (ADT)
- ADT consists of
  - Domain
    - Range of values that can be stored
  - Operations
    - What operations can be performed with that value
  - Axioms
    - How the operations which can be performed, interplay among themselves
  - Type is a precise characterization of structural or behavioral properties which a collection of all entities will share

# Typing

- Built-in (Primitive) Types
  - Int
  - Double
  - char
  - bool
- User-defined Types (UDT)
  - Complex
  - Vector
  - Employee
  - Executive
  - Leave
  - CasualLeave

# Type of Programming Languages

- A programming language may be
  - **Statically typed** – static binding or early binding [C]
    - Types are associated with **Variables** not **Values**
    - Variable has a type and it is defined while writing the programing
    - Compiler know the type of the variable at **compile-time**
    - Type of a variable always remains constant (check line 3-4)
    - Static type checking is the process of verifying the type safety of a program based on analysis of a program's text
    - Example: C, C++, Java, etc.

```
int iVal = 2;          // iVal is of type int
double dVal = 3.6;    // dVal is of type double
iVal = 3.6;           // Implicit conversion of double --> int
dVal = 2;             // Implicit conversion of int --> double
```

# Type of Programming Languages

- **Dynamically typed** – dynamic binding or late binding
  - Types are associated with **values not variables** [Python]
  - Type of a variable is associated with the **type of value** assigned to it and not type of the variable
  - Dynamic type checking is the process of verifying the type of program at **run-time**
  - Example: Python, (C++ and Java) via Polymorphism, etc.

```
iVal = 2      // iVal is of type int
dVal = 3.6    // dVal is of type double
iVal = 3.6    // iVal is of type double
dVal = 2      // dVal is of type int
```

# Type of Programming Languages

- Strongly Typed
  - Typing errors are prevented at runtime
  - Allow little implicit type conversion
  - Does not use static type checking
  - Compiler does not check or enforce type constraint
  - Example: Python, C++, Java
- Weakly Typed
  - Allow implicit type conversion
  - Example: C, some features of C++ and Java
- Untyped
  - There is no type checking and any type conversion reqd. is explicit
  - Example: Assembly level language

# Polymorphism

- Polymorphism exists when dynamic typing and inheritance interact
- A single name (such as variable) may denote multiple objects of different classes that are related by some common superclass
- dynamic typing:
  - X = 2;
  - X = 2.5;
- Polymorphism is a very important tool to perform abstraction in an OOP.

# Signatures



# Signatures

- In any programming language, a **signature** is what distinguishes one function or method from another
- In C, every function has to have a different name
- In Java, two methods have to differ in their *names* or in the *number* or *types* of their parameters
  - `foo(int i)` and `foo(int i, int j)` are different
  - `foo(int i)` and `foo(double k)` are different
  - `foo(int i, double d)` and `foo(double d, int i)` are different
- In C++, the signature **also includes the *return type***
  - But not in Java!

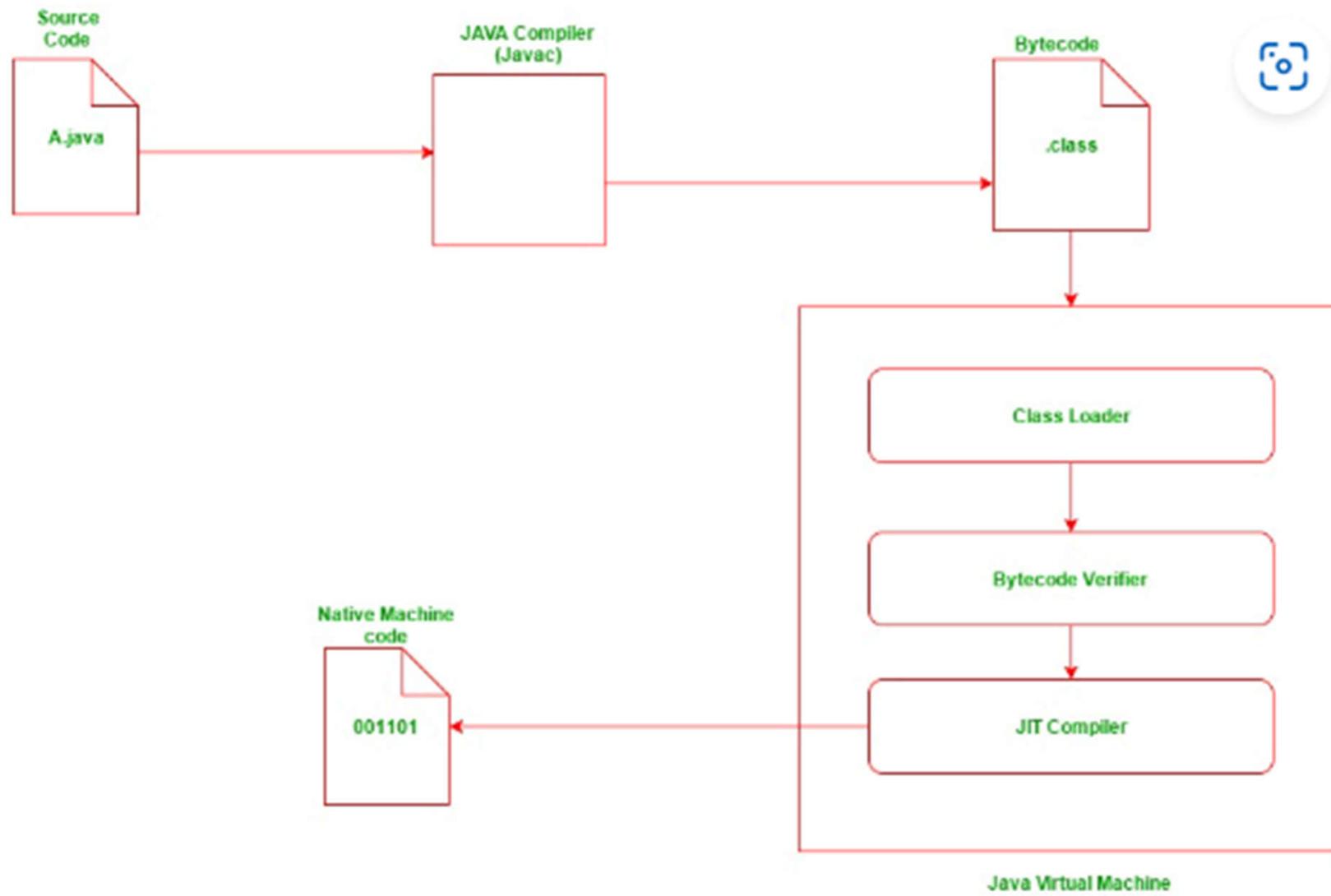
# Polymorphism

- Polymorphism means *many* (poly) *shapes* (morph)
- In Java, polymorphism refers to the fact that you can have multiple methods with the same name in the same class
- A method in Java can behave in multiple ways, thus performing different sub-tasks
- So the method posses **different behavior at different situations** and a programmer can make use of this feature.

# Polymorphism

- This feature is known as Polymorphism
- It allows to perform same task in different ways
- Java performs polymorphism in two ways:
  - Compile time polymorphism
  - Run time polymorphism

# Java Program Execution Steps



# Polymorphism

- Compile time polymorphism (Static/Early binding)
  - Function call to such method is resolved at compile time
  - Performed by Method Overloading
  - Also known as Static polymorphism
- Run time polymorphism (Dynamic method dispatch)
  - Performed by Method Overriding
  - Function call to such method is resolved at runtime
  - Replaces an inherited method with another having the same signature.

# Overloading

```
class Test {  
    public static void main(String args[]) {  
        void myPrint(5);  
        void myPrint(5.0);  
    }  
  
    static void myPrint(int i) {  
        System.out.println("int i = " + i);  
    }  
  
    static void myPrint(double d) { // same name, different parameters  
        System.out.println("double d = " + d);  
    }  
}  
  
int i = 5  
double d = 5.0
```

# Why overload a method?

- So you can use the same names for methods that do essentially the same thing
  - Example: `println(int)`, `println(double)`, `println(boolean)`, `println(String)`, etc.
- So you can supply defaults for the parameters:

```
int increment(int amount) {  
    count = count + amount;  
    return count;  
}
```

```
int increment() {  
    return increment(1);  
}
```

- Notice that one method can call another of the same name

# DRY (Don't Repeat Yourself)

- When you overload a method with another, very similar method, only one of them should do most of the work:

```
void dict_Example( ) {  
    System.out.println("first = " + first + ", last = " + last);  
    for (int i = first; i <= last; i++) {  
        System.out.print(dictionary[i] + " ");  
    }  
    System.out.println();
```

```
void dict_Example(String s) {  
    System.out.println("At checkpoint " + s + ":");  
    dict_Example();  
}
```

# Legal assignments

```
class Test {  
    public static void main(String args[]) {  
        double d;  
        int i;  
        d = 5;          // legal  
        i = 3.5;        // illegal  
        i = (int) 3.5;  // legal  
    }  
}
```

- Narrowing is legal
- Widening is illegal (unless you cast)

# Legal method calls

```
class Test {  
    public static void main(String args[]) {  
        myPrint(5);  
    }  
  
    static void myPrint(double d) {  
        System.out.println("Output: " + d);  
    }  
}
```

Output 5.0

- Legal because parameter transmission is equivalent to assignment
- `myPrint(5)` is like `double d = 5;`  
`System.out.println(d);`

# Illegal method calls

```
class Test {  
    public static void main(String args[]) {  
        myPrint(5.2);  
    }  
  
    static void myPrint(int i) {  
        System.out.println(i);  
    }  
}
```

myPrint(int) in Test cannot be applied to (double)

- Illegal because parameter transmission is equivalent to assignment
- myPrint(5.2) is like int i = 5.2; System.out.println(i);

# Method Overloading

- It is a technique in which a single class contains
  - Multiple methods
  - Of same name
  - But different parameter list
  - For distinguishing among themselves
  - Object of the class is used to call each method.

```
class test{  
    void display() {}  
    void display(int x) {}  
    void display(int x, int y) {}  
}
```

# Constructor Overloading

- It is a technique in which a single class contains
  - Multiple constructors
  - Of same name
  - But different parameter list
  - For distinguishing among themselves
  - Object of the class is not used to call the constructors.

```
class test{  
    test( ) {}  
    test(int x) {}  
    test(int x, int y) {}  
}
```

# Java uses the most specific method

```
class Test {  
    public static void main(String args[]) {  
        myPrint(5);  
        myPrint(5.0);  
    }  
    static void myPrint(double d) {  
        System.out.println("double: " + d);  
    }  
    static void myPrint(int i) {  
        System.out.println("int: " + i);  
    }  
}
```

int: 5  
double: 5.0

# Operator Overloading

- Methods and Operators can be overloaded;
  - Method overloading (includes constructors)
  - Operator overloading
- Java does not support Operator Overloading
- C++ does support Operator Overloading along with Method/Function overloading.

# Operator Overloading

- Operators are overloaded to give user defined meaning to it
- Examples:  $t1 / t2$  or  $t1 == t2$
- Overloaded operator is used to perform different operations on variables
- Example: ‘+’ operator can be overloaded to perform addition on various data types, like for Integer, String concatenation, etc.

# Operator Overloading - Example

The diagram shows the expression `cout << "This is test string";`. Three annotations with arrows point to different parts of the expression:

- An arrow points from the text "object of **ostream** class" to the identifier `cout`.
- An arrow points from the text "string" to the string literal `"This is test string"`.
- An arrow points from the text "overloaded insertion operator" to the operator symbol `<<`.

```
cout << "This is test string";
```

# Operator Overloading

- Almost all operators can be overloaded
- Few operators cannot be overloaded
- List of operators which cannot be overloaded are;
  - scope operator - `::`
  - **sizeof**
  - member selector - `.`
  - member pointer selector - `*`
  - ternary operator - `?:`

# Operator Overloading - Syntax

Keyword      Operator to be overloaded

```
ReturnType classname :: Operator OperatorSymbol (argument list)
```

```
{
```

```
    // Function body
```

```
}
```

# Operator Overloading - Restrictions

- Precedence and Associativity of an operator cannot be changed.
- Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure. You cannot change how integers are added.

# / Operator Overloading - Example

```
class test
{
    int n;
public:
    test( )
    {
        cin >> n;
    }

void operator/(test);

};
```

```
void test::operator/(test t2)
{
    cout << n / t2.n;
}

int main( )
{
    test t1, t2;
    t1 / t2;
    return 0;
}
```

OUTPUT:

```
10 <= t1.n
5   <= t2.n
2
```

# << Operator Overloading - Example

```
class test {  
    int n;  
public:  
    test( ) {  
        cin>>n;  
    }  
    void operator/(test);  
    friend ostream& operator<< ( ostream&, test& );  
};  
  
void test::operator/(test t2) {  
    cout << n/t2.n;  
}
```

# << Operator Overloading - Example

```
ostream& operator<< ( ostream &out, test &t1 )  
{  
    out << endl << "Value is " << t1.n;  
    return out;  
}  
  
int main( )  
{  
    test t1,t2;  
    t1 / t2;  
    cout << t1;  
    return 0;  
}
```

# == Operator Overloading - Example

```
class test {  
    int n;  
public:  
test( ) {  
    cin>>n;  
}  
void operator/(test);  
friend bool operator== ( test, test );  
};  
  
void test::operator/(test t2) {  
    cout << n/t2.n;  
}
```

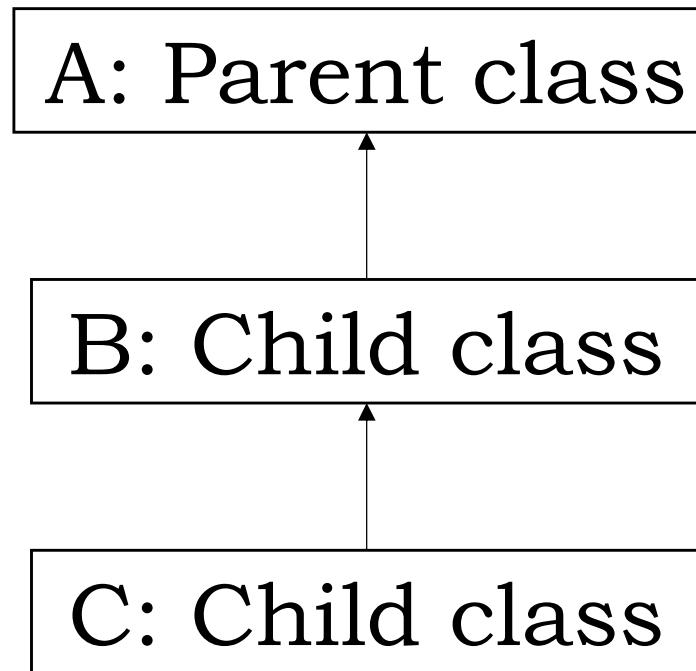
# == Operator Overloading - Example

```
bool operator== ( test t1, test t2 )
{
    return ( t1.n == t2.n );
}

int main( )
{
    test t1,t2;
    t1/t2;
    cout << endl << ( t1 == t2 );
    return 0;
}
```

# Multilevel Inheritance

- A child class derives member variables and methods from another derived class.



# Method Overriding

- If subclass (child class) has the same method as declared in the parent class, it is known as method overriding
- If subclass provides specific implementation of the method that has been provided by one of its parent class, it is known as method overriding
- Advantage:
  - Provide specific implementation of a method that is already provided by its super class
- Rules:
  - Same name as method in parent class
  - Same parameter as in the parent class
  - Must be in a IS-A relationship (inheritance)

# Method Overriding - Example

```
class Bank{  
int getRateOfInterest( )  
{ return 0; }  
}
```

```
class SBI extends Bank{  
int getRateOfInterest( )  
{ return 8; }  
}
```

```
class ICICI extends Bank{  
int getRateOfInterest( )  
{ return 7; }  
}
```

```
class AXIS extends Bank{  
int getRateOfInterest( )  
{ return 9; }  
}  
  
public class Method_OVERRIDING{  
psvm(S a[]){  
  
SBI s=new SBI( );  
ICICI i=new ICICI( );  
AXIS a=new AXIS( );  
S.O.P("SBI"+ s.getRateOfInterest( ));  
S.O.P("ICICI"+i.getRateOfInterest());  
S.O.P("AXIS"+a.getRateOfInterest());  
  
}  
}
```

# Method Overloading vs Overriding

<b>Method Overloading</b>	<b>Method Overriding</b>
Used to increase readability of a program	Used to provide the specific implementation of the method that is already provided by its super class
Always performed within a class	Occurs in two classes that have a IS-A relationship
Parameters of two overloaded methods should be different.	Two overridden methods should have same parameters.

# Access Specifiers

- Types of Access Specifiers:
  - Public
  - Default (Friendly)
  - Protected
  - Private

# Access Specifiers or Visibility Modes

- Provides a restriction over the scope of a variable/entity of a program
- Scope is applicable with respect to the following;
  - Class
  - Sub-class
  - Package
    - It is a group of similar types of classes, interfaces, etc.

# Public Access Specifier

- Public attributes/methods are accessible from anywhere in the program
- All sub classes can access public members
- Classes in other packages can also access public members.

# Protected Access Specifier

## ■ WHO CAN ACCESS

1. All members within the same class
2. All members within sub-classes
3. Sub-classes present in the same package
4. Classes of same package which are not related
5. Sub-classes present in other packages.

## ■ WHO CANNOT ACCESS

6. Classes in other packages, which are not sub-classes.

# Default (Friendly) Access Specifier

## ■ WHO CAN ACCESS

1. All members within the same class
2. All members within sub-classes
3. Sub-classes present in the same package
4. Classes of same package which are not related.

## ■ WHO CANNOT ACCESS

5. Sub-classes present in other packages
6. Classes in other packages, which are not sub-classes.

# Private Protected Access Specifier

## ■ WHO CAN ACCESS

1. All members within the same class
2. All members within sub-classes
3. Sub-classes present in the same package
5. Sub-classes present in other packages

## ■ WHO CANNOT ACCESS

4. Classes of same package which are not related
6. Classes in other packages, which are not sub-classes.

□ **DEPRECATED ... NOT USED NOW !!!**

# Private Access Specifier

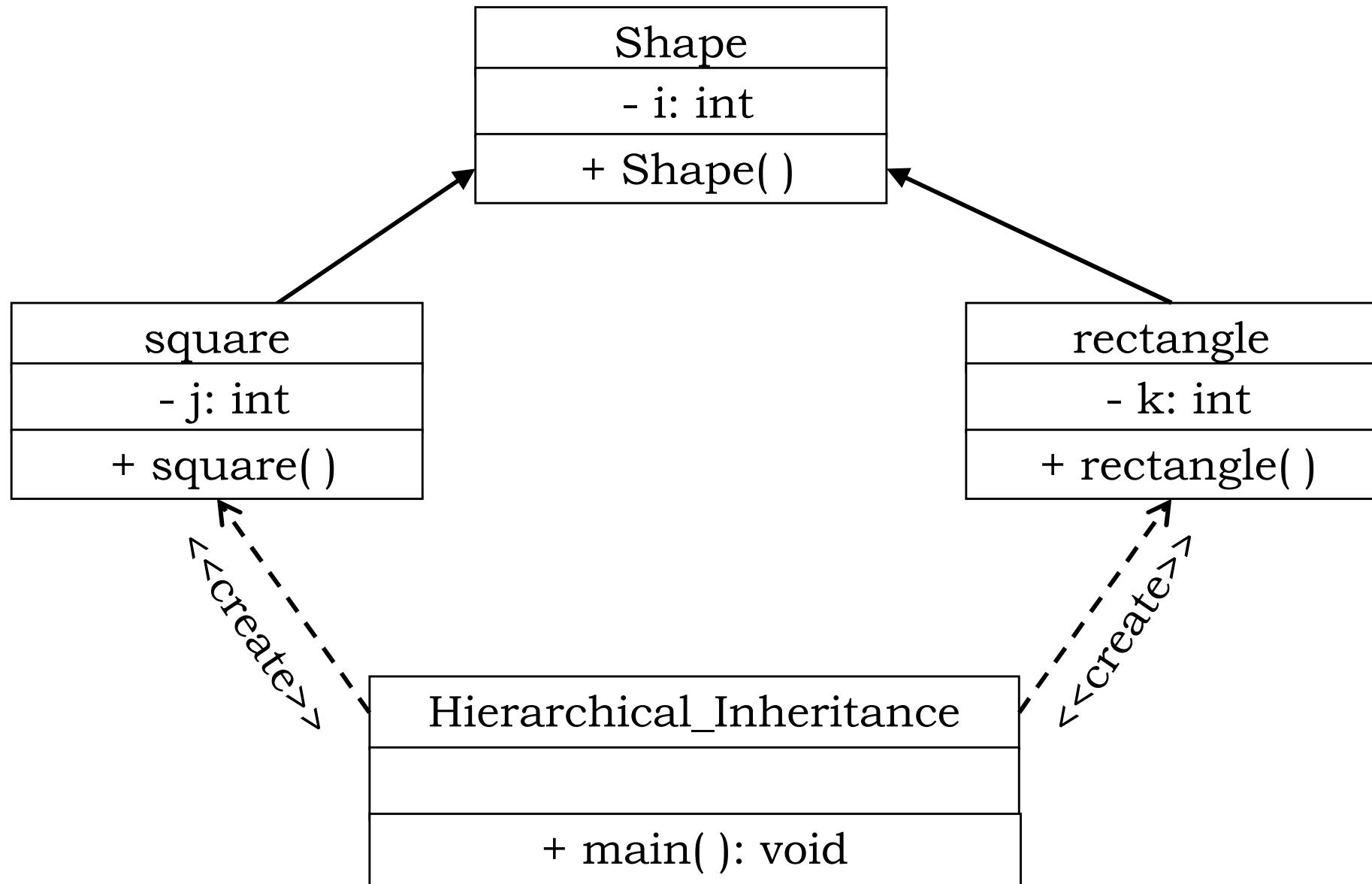
## ■ WHO CAN ACCESS

1. All members within the same class

## ■ WHO CANNOT ACCESS

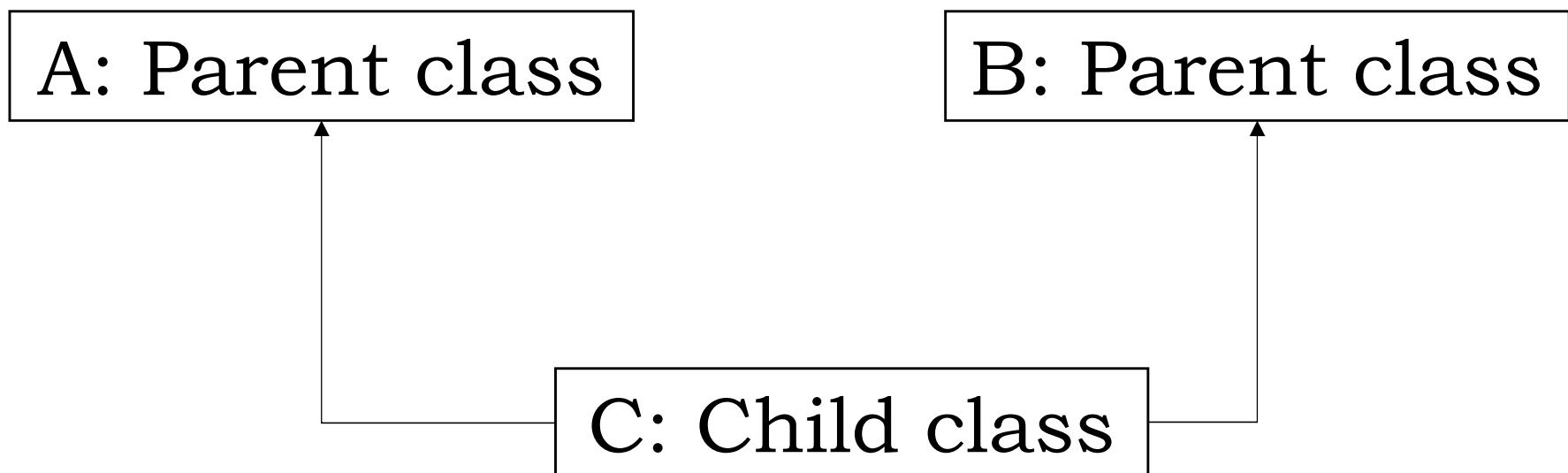
2. All members within sub-classes
3. Sub-classes present in the same package
4. Classes of same package which are not related
5. Sub-classes present in other packages
6. Classes in other packages, which are not sub-classes.

# Hierarchical Inheritance – Class Diagram



# Multiple Inheritance

- A child class derives member variables and functions from multiple parent classes.



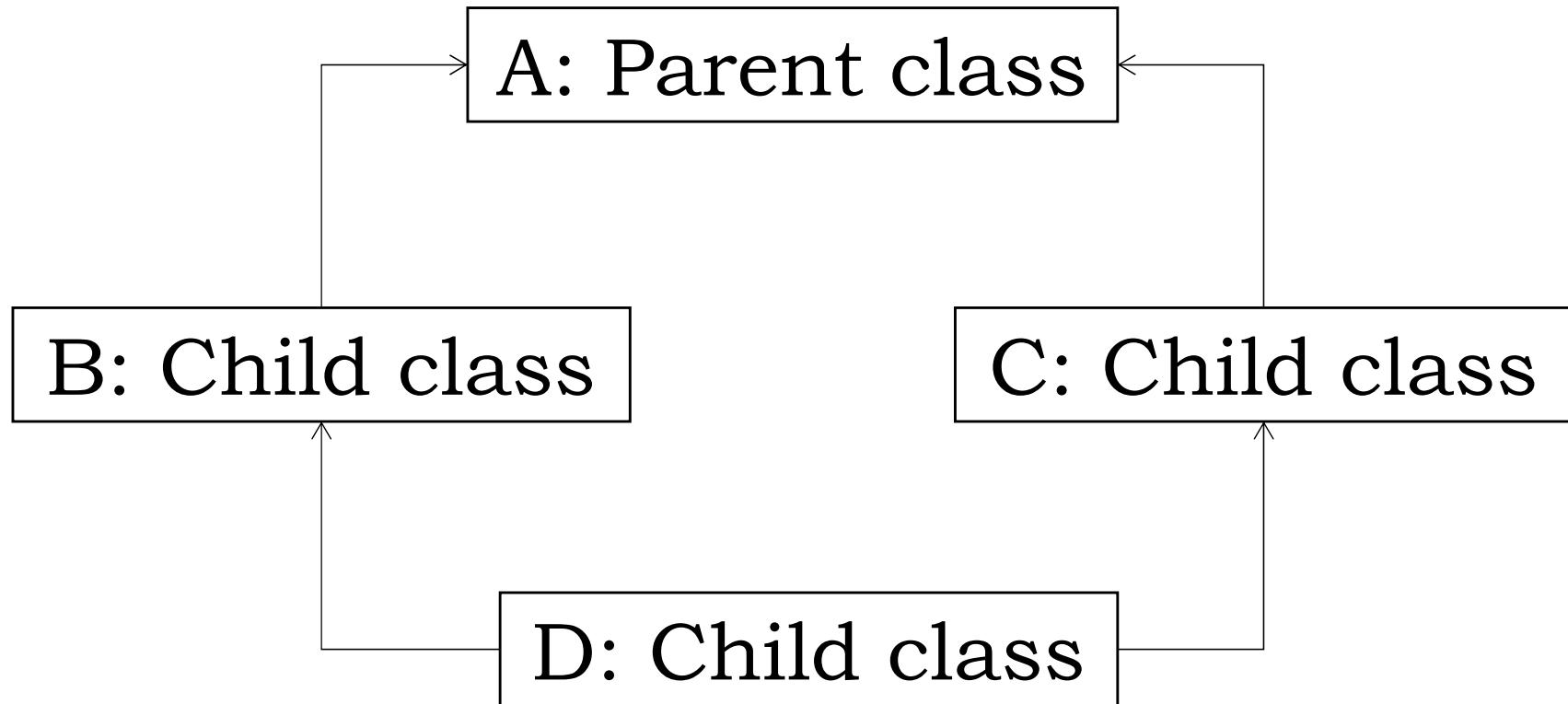
# Multiple Inheritance – Example C++

```
class One {  
    protected:  
        int x;  
    public:  
        One( ) { x = 5; }  
        void disp( )  
        {  
            cout << "IIIT" << endl;  
        }  
};  
  
class Two {  
    protected:  
        int y;  
    public:  
        Two( ) { y = 50; }  
};
```

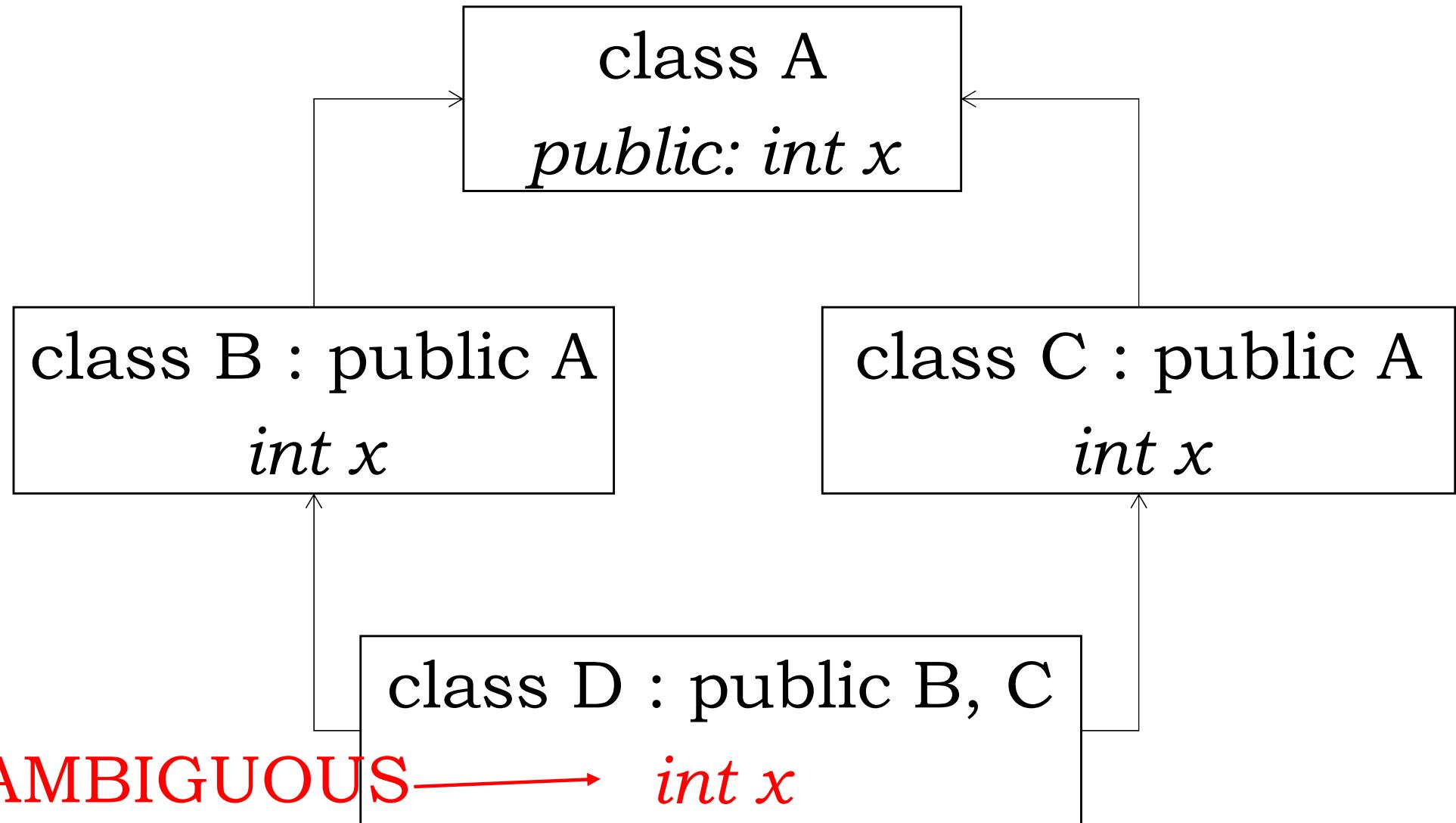
```
void show( ) {  
    cout << "Vadodara" << endl;  
}  
};  
  
class Three : public One, public Two  
{  
    public:  
        void printValues( ) {  
            cout << x + y << endl;  
        }  
};  
  
int main( ) {  
    Three t1;  
    t1.printValues( );  
t1.disp( ); t1.show( );  
}
```

# Hybrid Inheritance

- Combination of two or more types of inheritances.



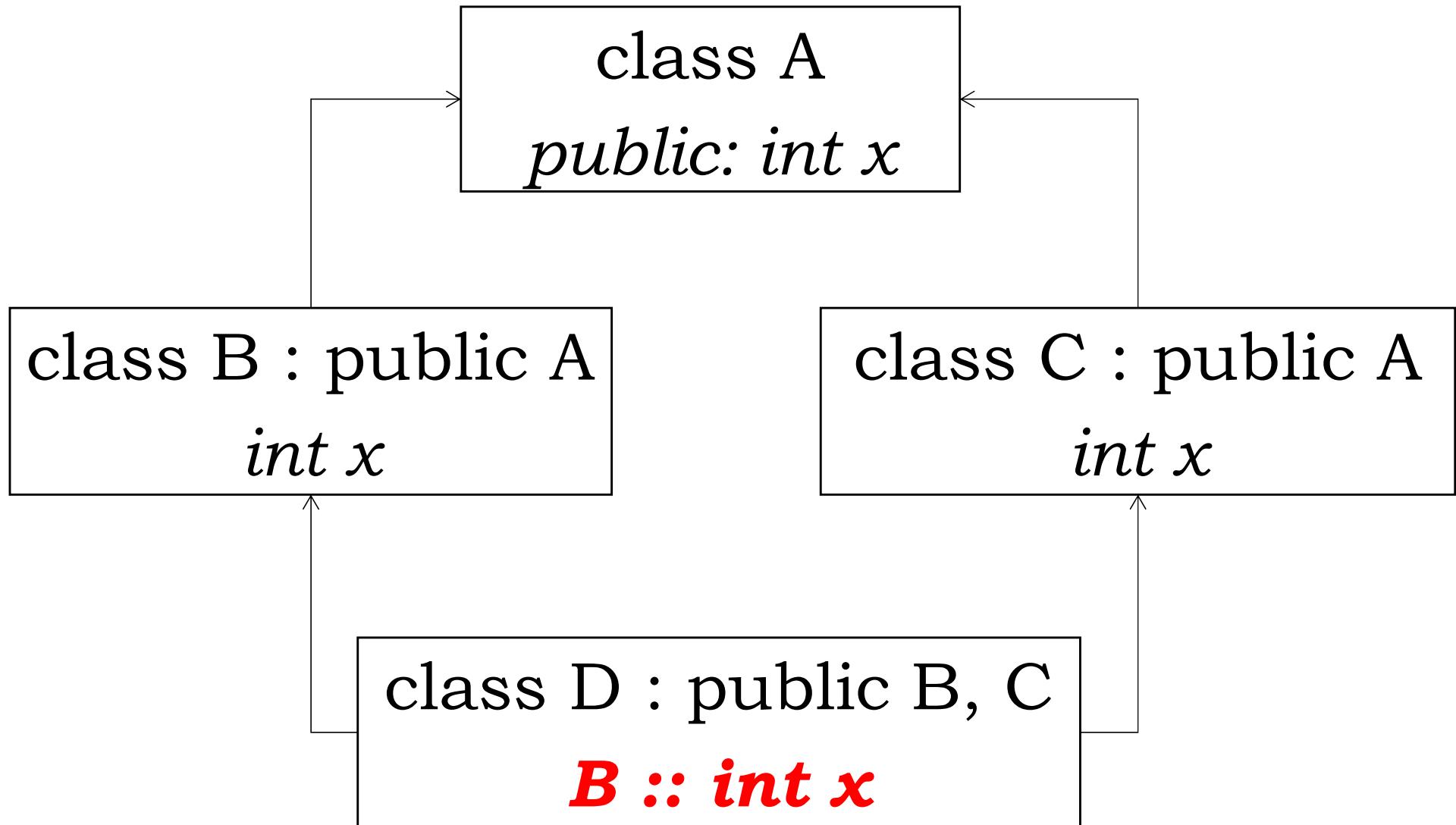
# Hybrid Inheritance – Problem



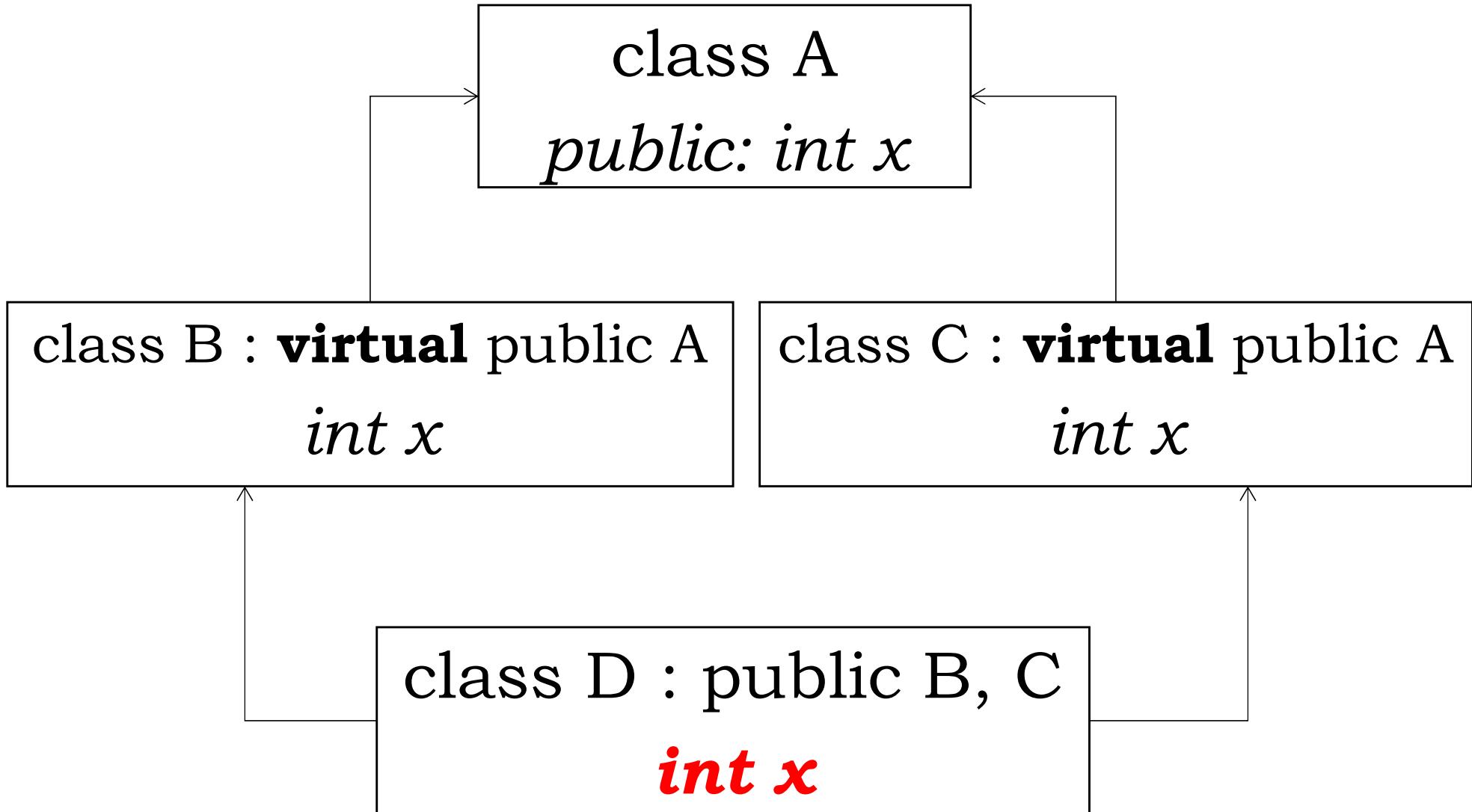
# Hybrid Inheritance – Solution in C++

- Two ways to counter this problem in C++:
  1. Scope resolution operator
  2. Virtual Base class

# Hybrid Inheritance – Solution-1



# Virtual Base Class



# Constructor calling - Inheritance

- Base class constructor is called by default
- First the Base class constructor is executed, then the derived class constructor executes.

# Constructor calling - Inheritance

```
class First
{
    First( )
    {
        System.out.println("IIIT");
    }
}

class Second extends First
{
    Second( )
    {
        System.out.println("Vadodara");
    }
}
```

```
public class
ConstructorCalling_Inheritance
{
    public static void main(String args[])
    {
        Second S = new Second();
    }
}
```

## **OUTPUT:**

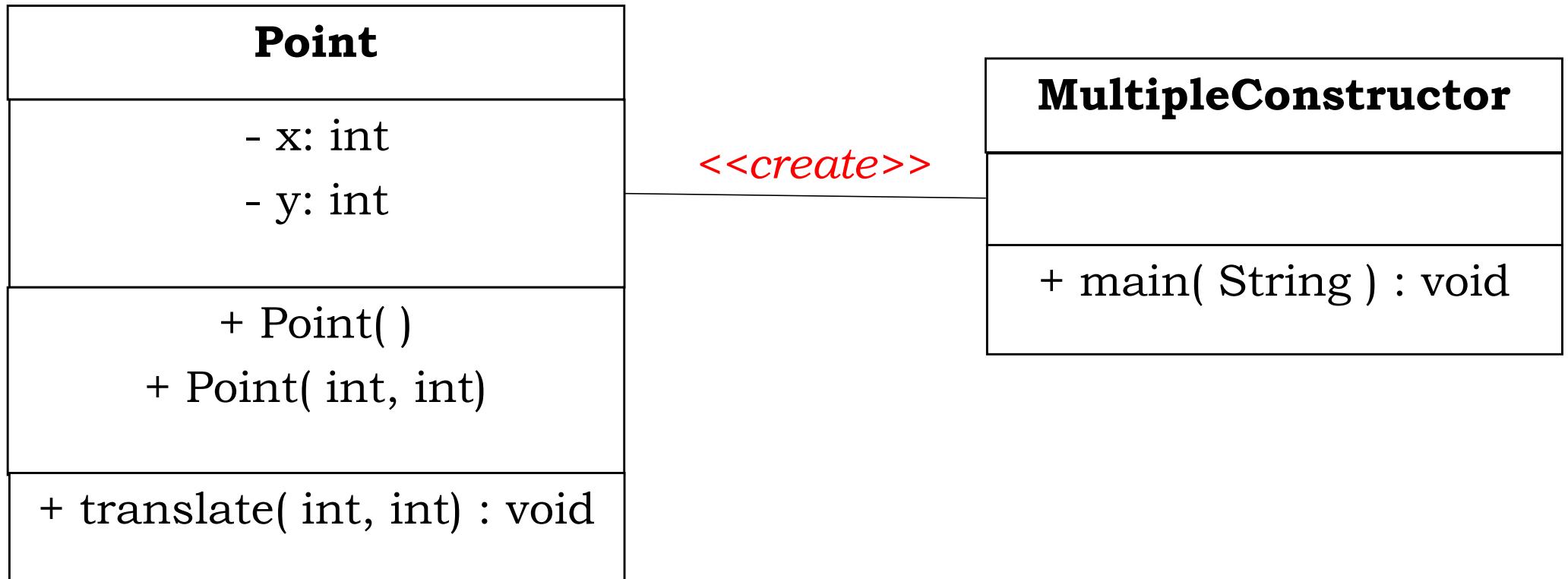
IIIT  
Vadodara

# Program to Class diagram

```
public class Point {  
  
    int x;    int y;  
    public Point( )  
    {  
        x = 0;  
        y = 0;  
    }  
public Point(int initialX, int initialY)  
{  
    x = initialX;  
    y = initialY;  
}  
  
public void translate(int dx, int dy)  
{  
    x = x + dx;  
    y = y + dy;  
}  
}
```

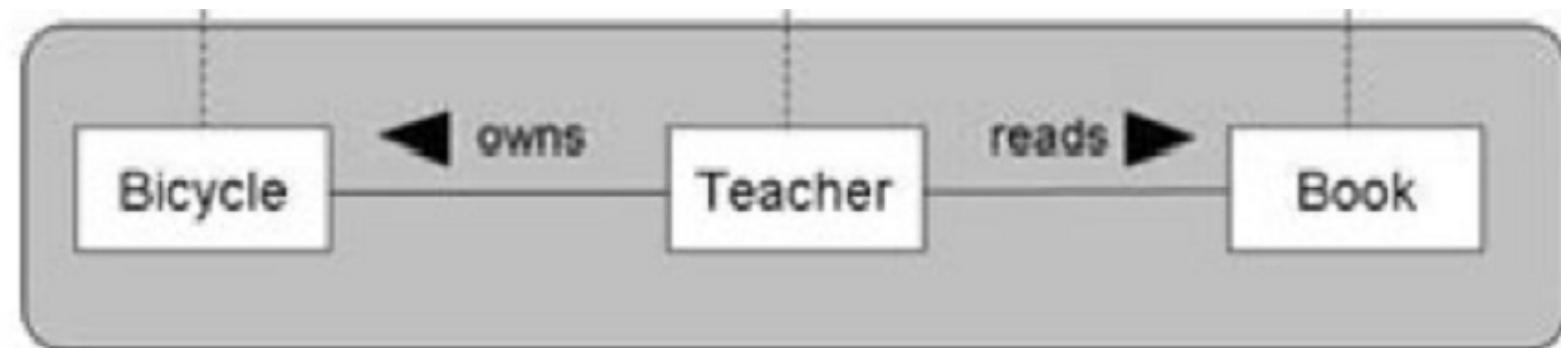
```
public class MultipleConstructor  
{  
    public static void main(String[]  
        args)  
    {  
Point p1 = new Point(5, 2);  
Point p2 = new Point(4, 3);  
Point p3 = new Point();  
  
    p2.translate(2, 4);  
  
    }  
}
```

# Program to Class diagram



# Class Diagrams

- A Class defines the attributes and the methods.
- All objects of this class (instances of this class) share the same behaviour, and have the same set of attributes (each object has its own set).
- Class diagrams provide a graphic notation for modeling classes and their relationships thereby describing possible objects.



# Class Diagrams

Class
+ attr1 : int
+ attr2 : string
+ operation1(p : bool) : double
# operation2()

# Problem #1: 2D Geometric Objects

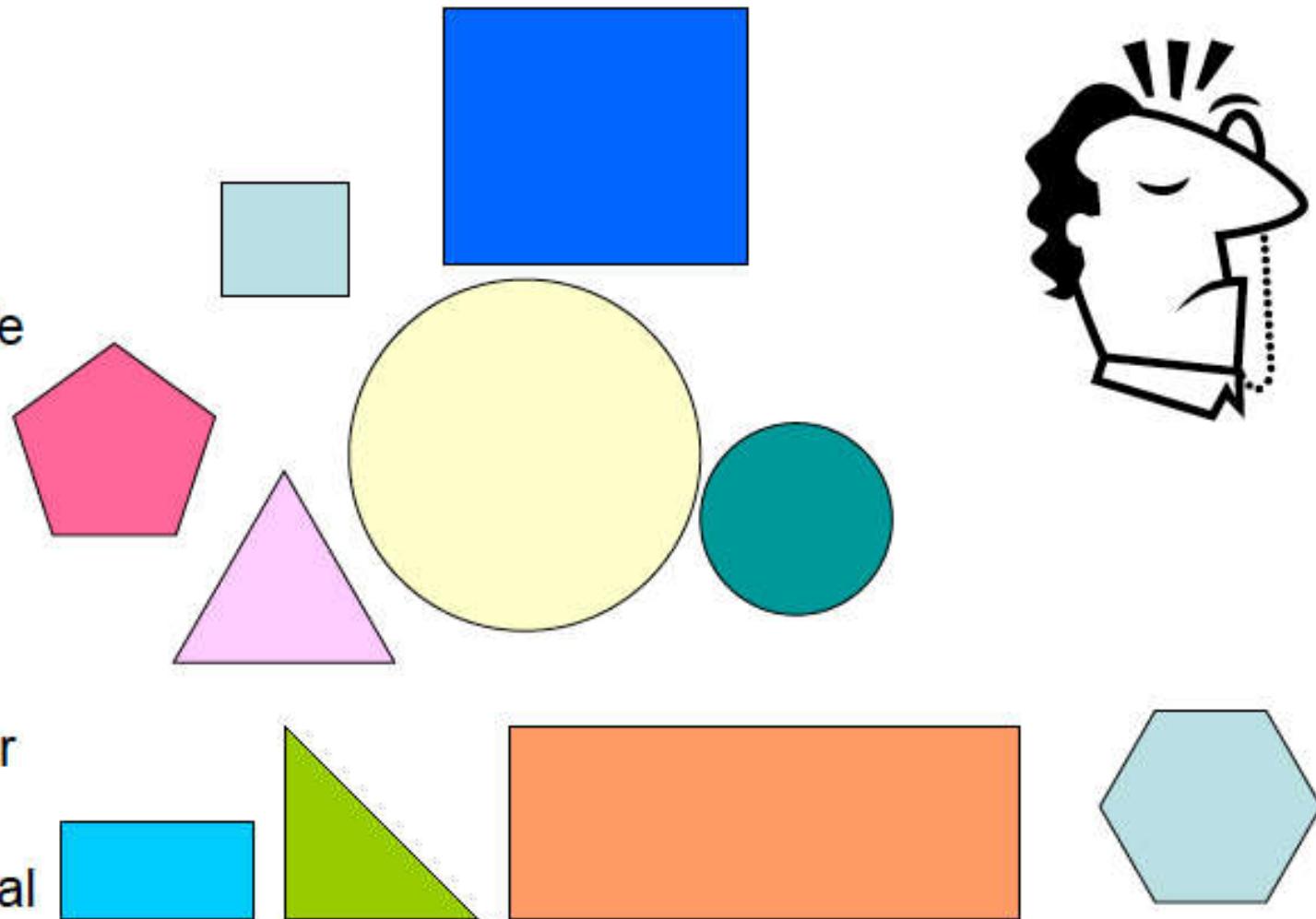
- Perform certain tasks on 2D shapes for example on **Polygons** and **Circles**
- A polygon may be triangle, rectangle, and square
- Shapes can be **Drawn** and **Color** can be assigned to the shapes
- The shape can be **Moved**
- Determine **Perimeter** and **Area** of a given shape.  
Isosceles or Equilateral for a Triangle.

# What do you notice?

- Regular 2D **shapes** can be **polygons** or **circles**
- A polygon consists of a number of **points** (>2)
- A polygon may be a **triangle**, a **rectangle**, **square**
- The other object is **circle**
- We can assign a **color** to a shape and **draw** it
- The shape can be **moved** to a **position**
- We should be able to determine **perimeter** and **area** of a given shape
- In case of a triangle, we determine whether, it is **equilateral** or **isosceles** triangle.

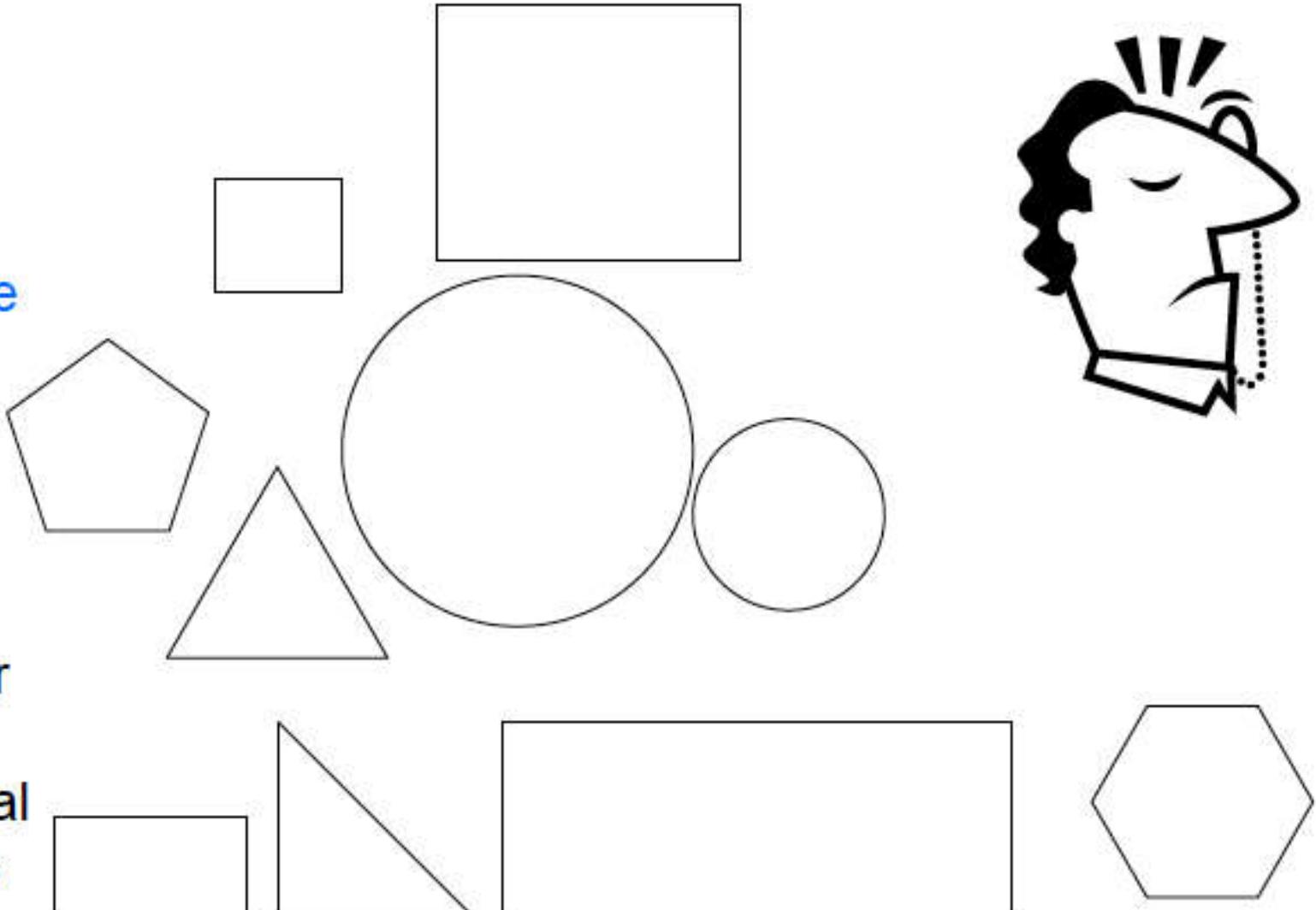
# What do you notice?

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



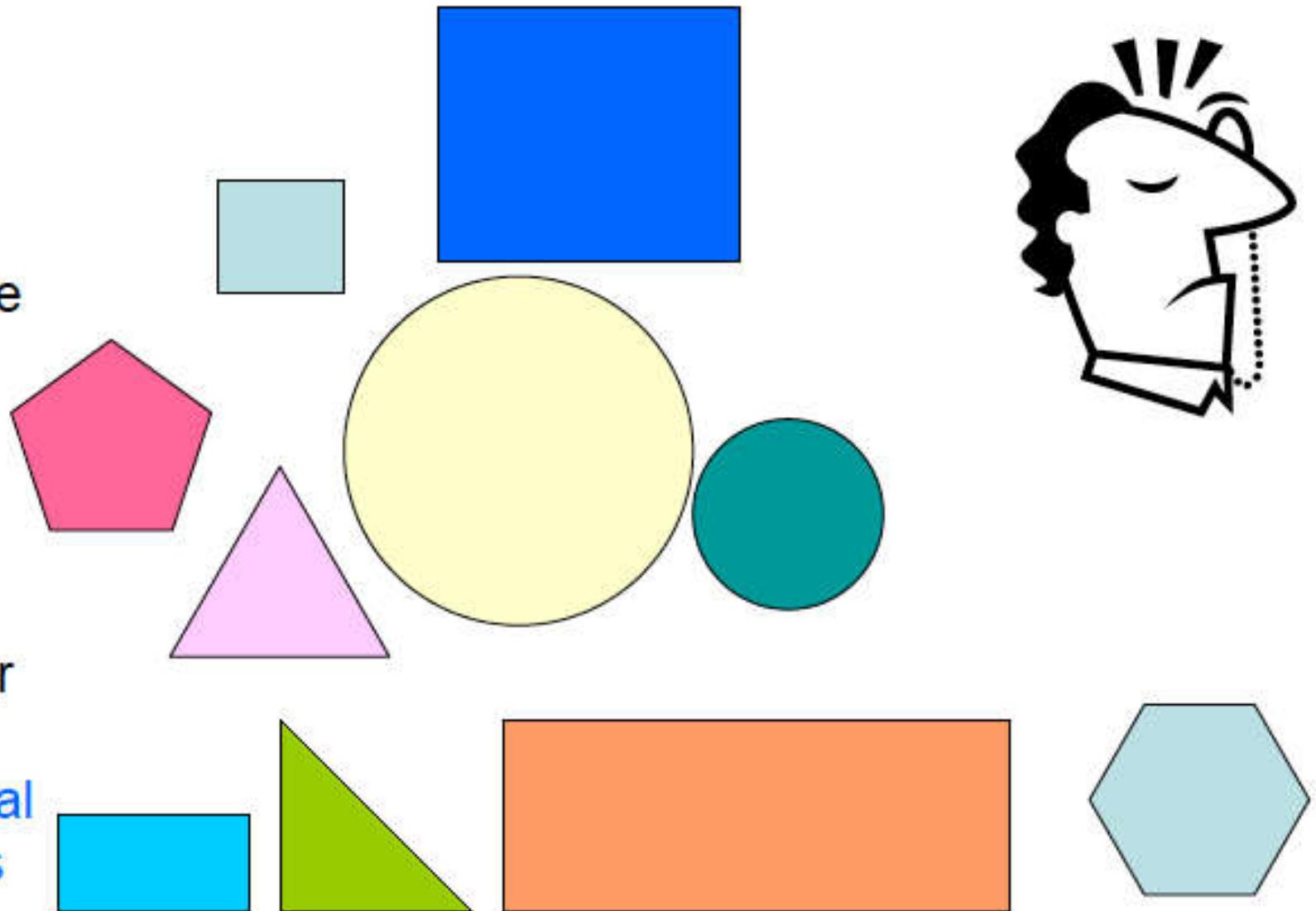
# What do you notice? Objects!

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



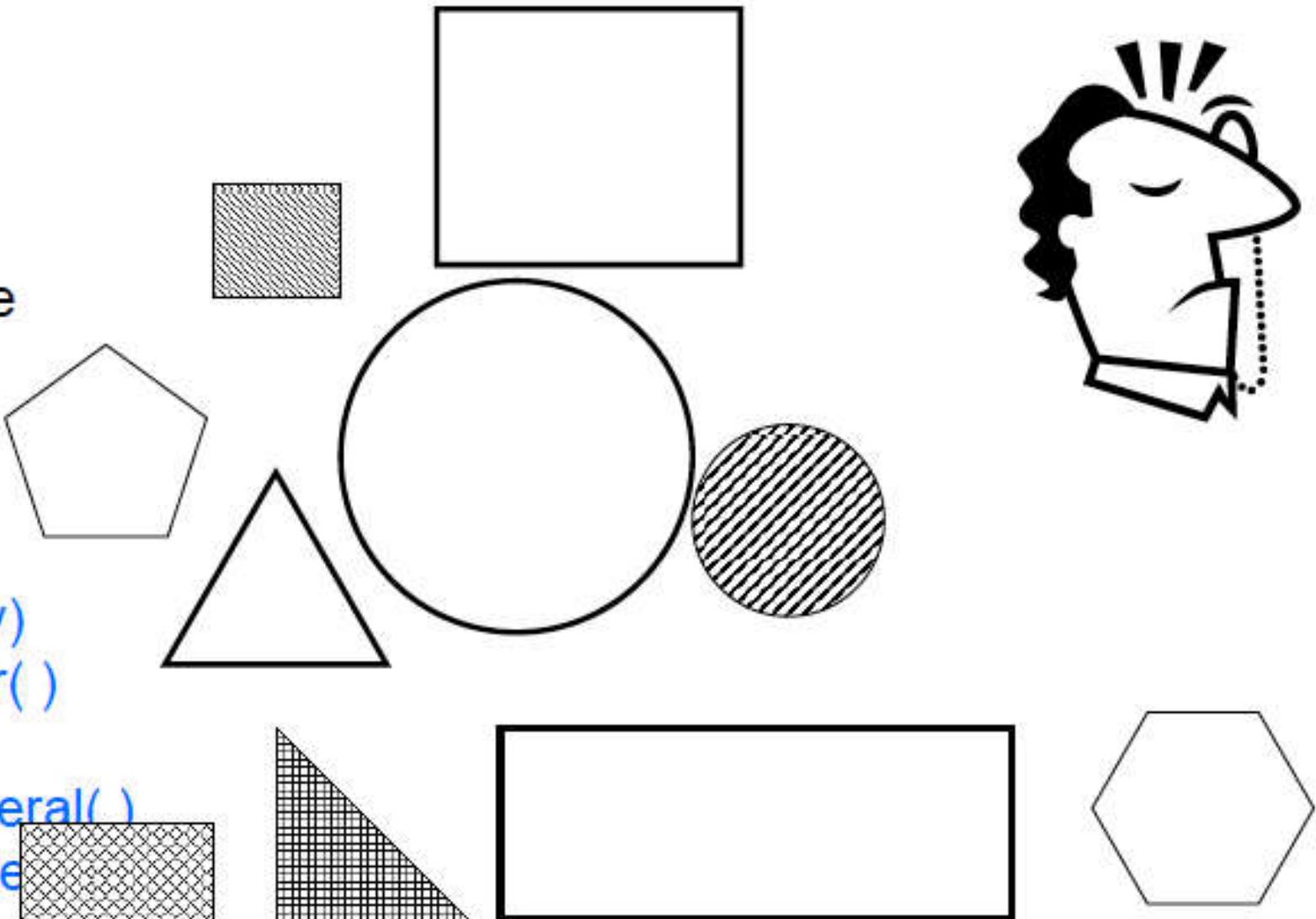
# What do you notice? Properties!

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



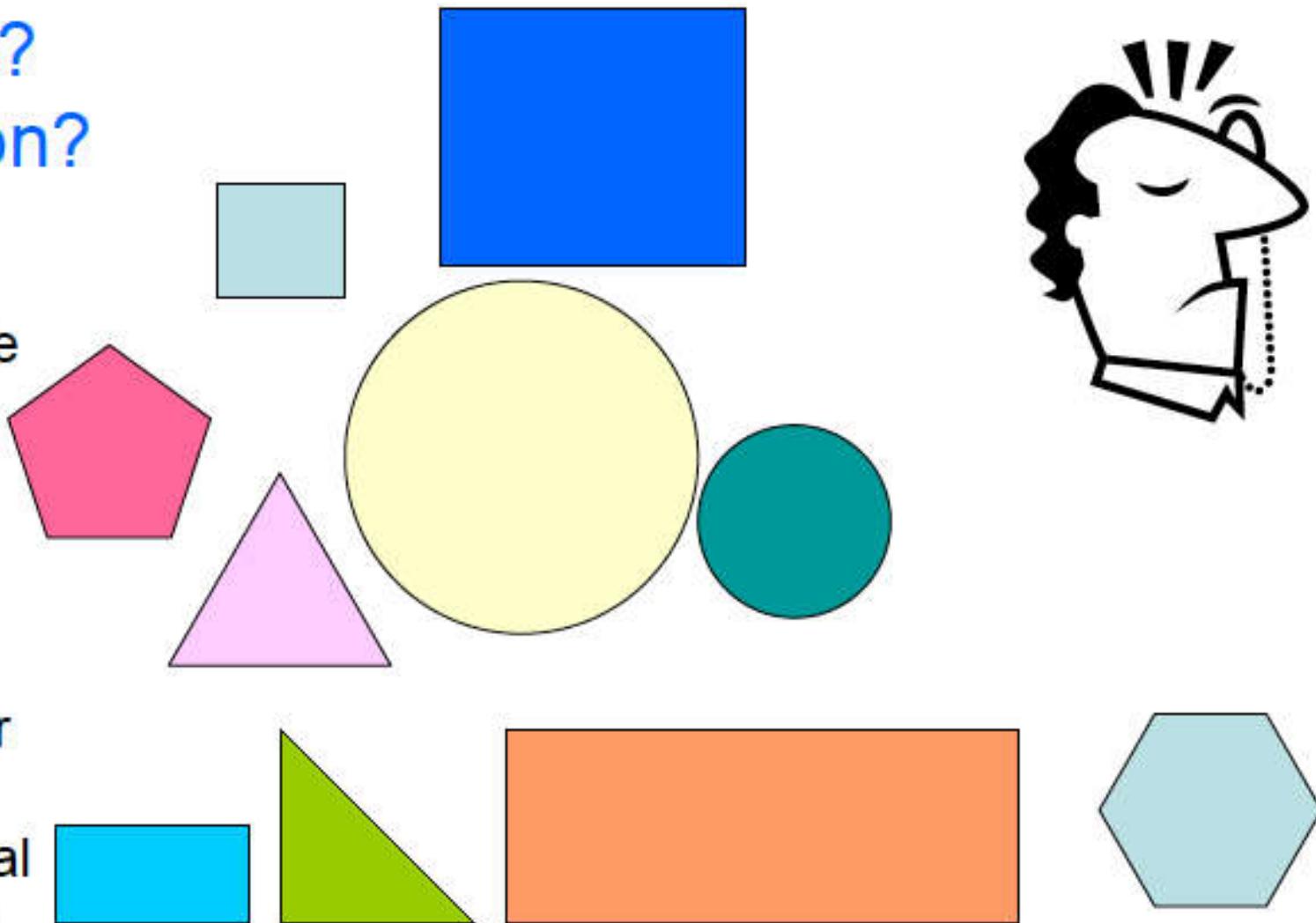
# What do you notice? Behavior

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw( )
- Move(x, y)
- Perimeter( )
- Area( )
- Is Equilateral( )
- Is Isosceles



# And What else?

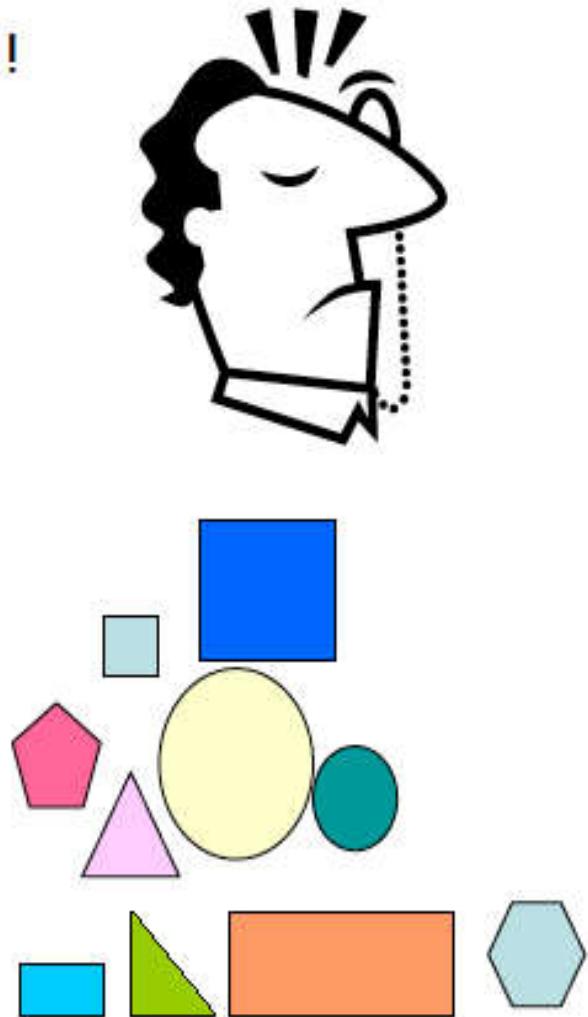
- Shape?
- Polygon?
- Circle
- Triangle
- Rectangle
- Square
- Color
- Point
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



# And What else?

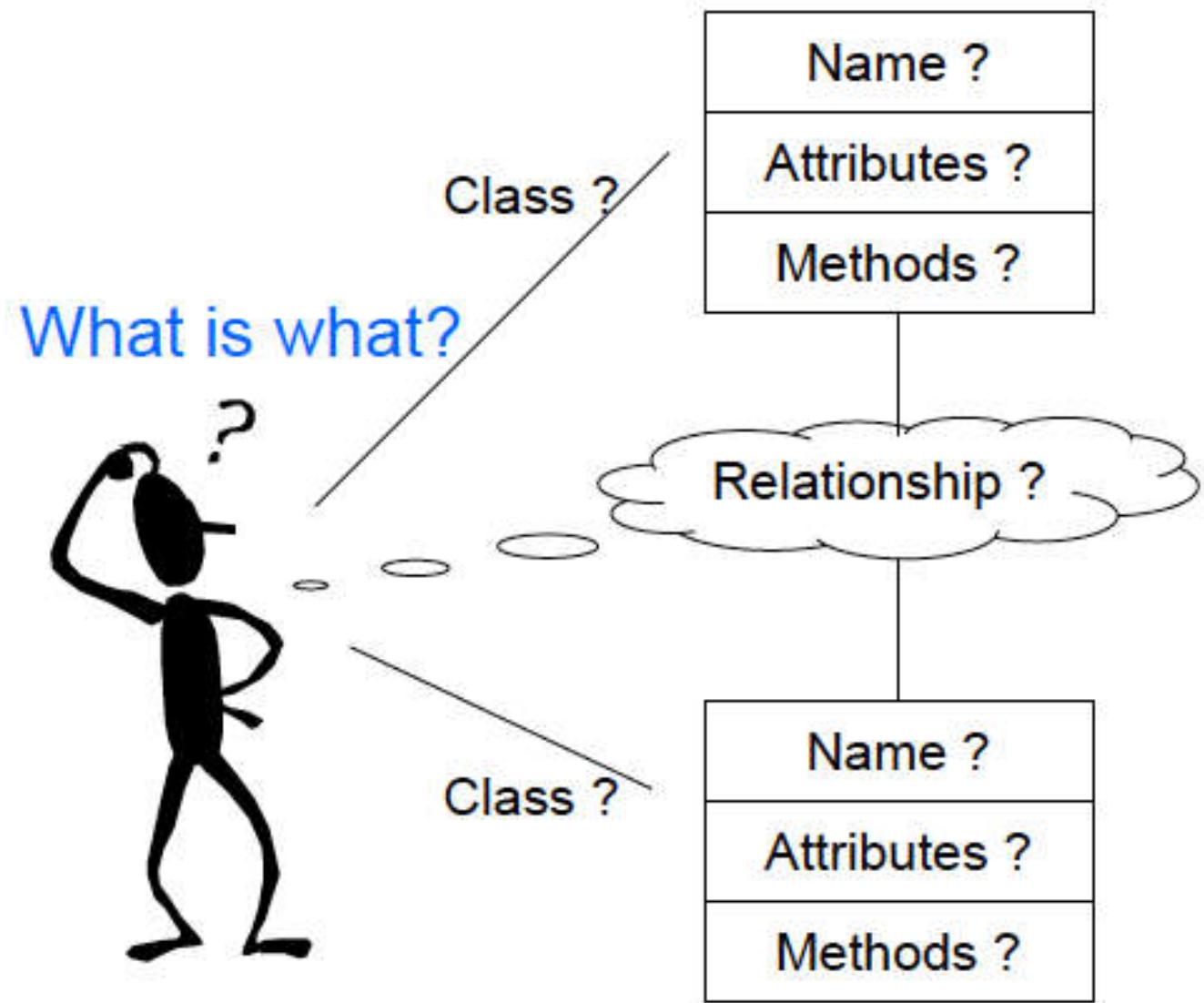
- Shape?
- Polygon?
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles

- All are **Shapes** !
  - Some of them are **Polygons** !
  - A “kind-of” relationship
  - Both are **abstract** !
- 
- A **Polygons** consists of a number of **points**
  - A “has-a” or “part-of” relationship!



# Classes?

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



# Classes

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles

Circle
Attributes ?
Methods ?

Triangle
Attributes ?
Methods ?

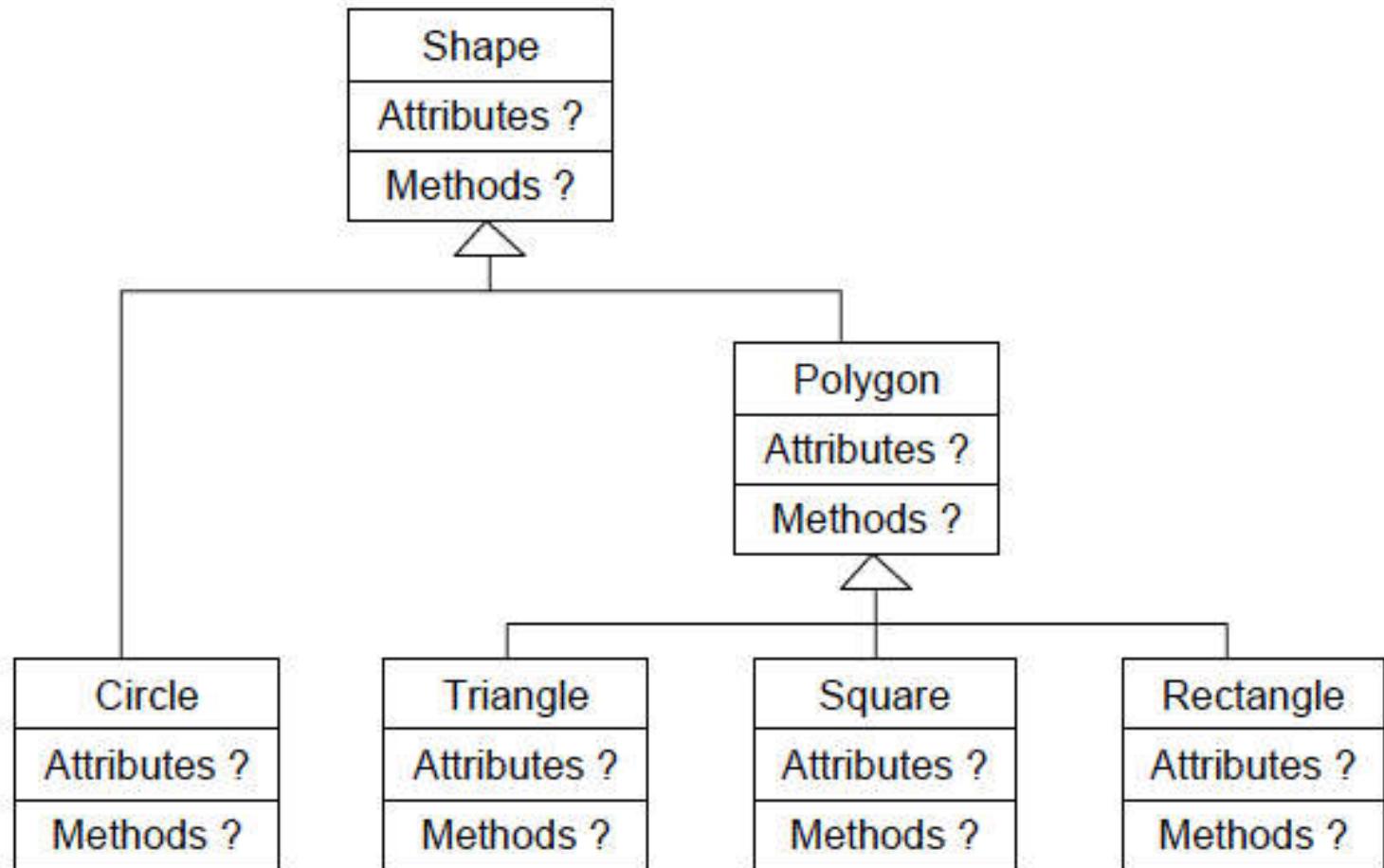
Square
Attributes ?
Methods ?

Rectangle
Attributes ?
Methods ?

One class per different type of objects

# Classes

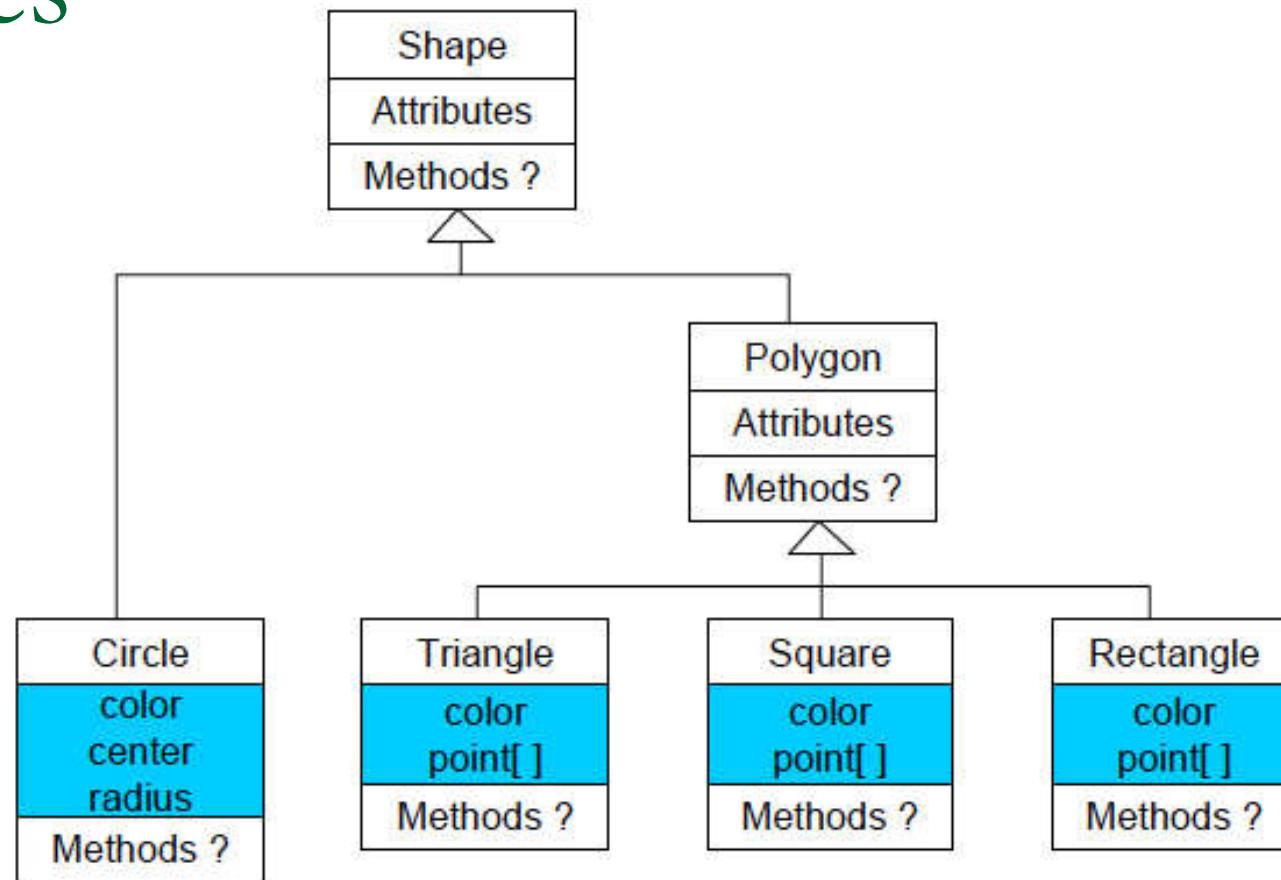
- Shape
- Polygon
  - Circle
  - Triangle
  - Rectangle
  - Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles



One class per different type of objects

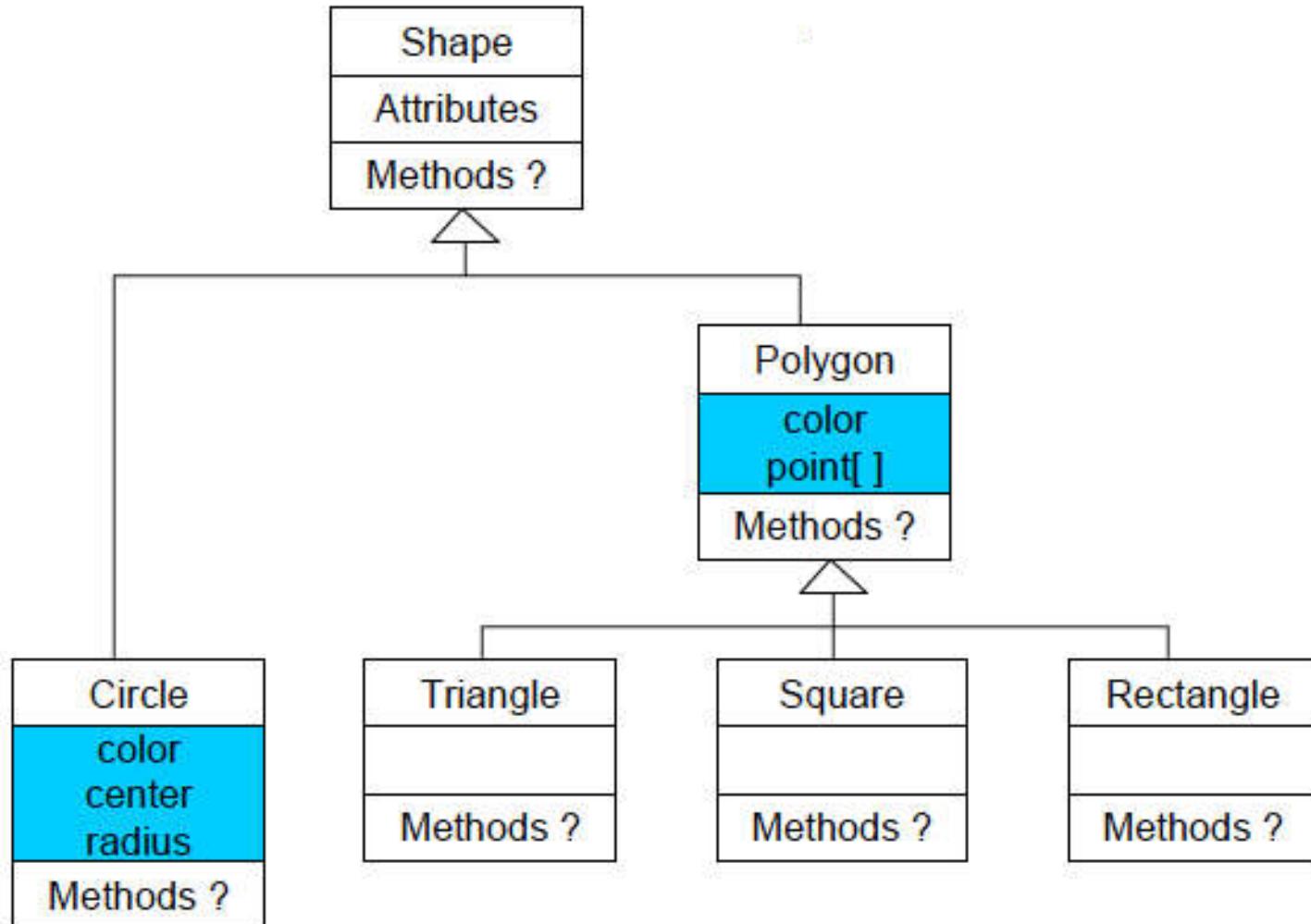
# Attributes

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles
- `getColor`
- `setColor`



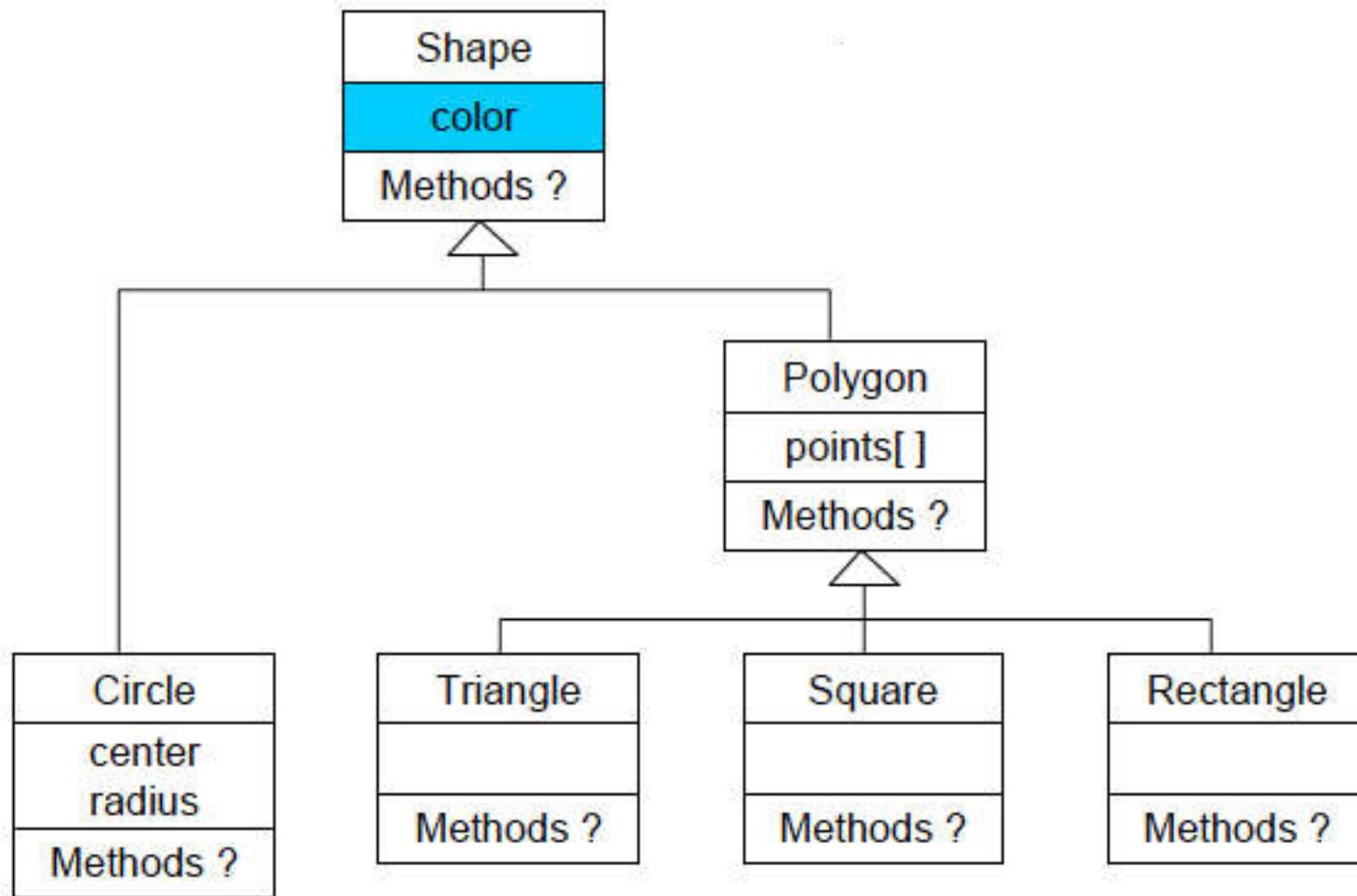
# Attributes

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles
- getColor
- setColor



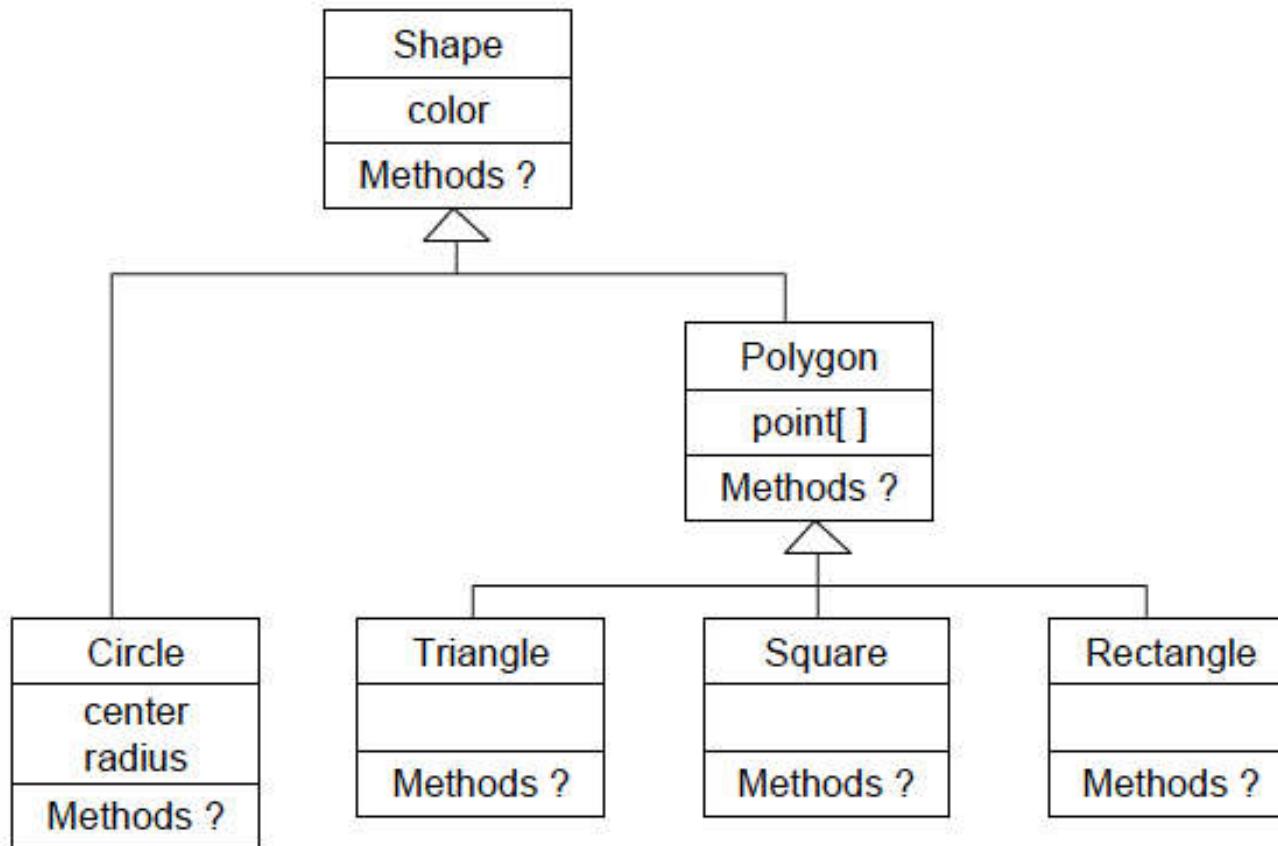
# Attributes

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles
- getColor
- setColor



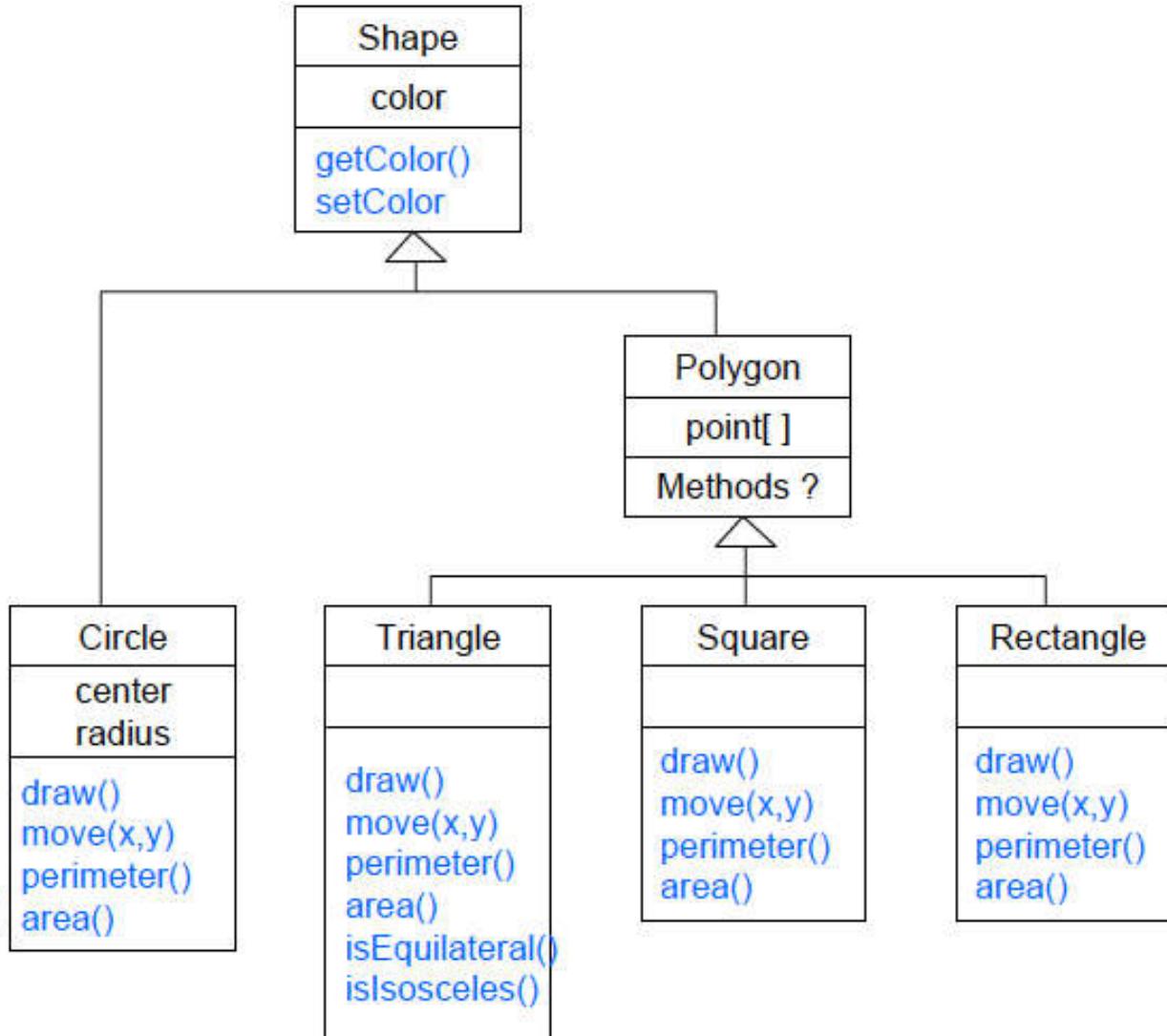
# Methods

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- Draw
- Move
- Perimeter
- Area
- Equilateral
- Isosceles
- getColor
- setColor



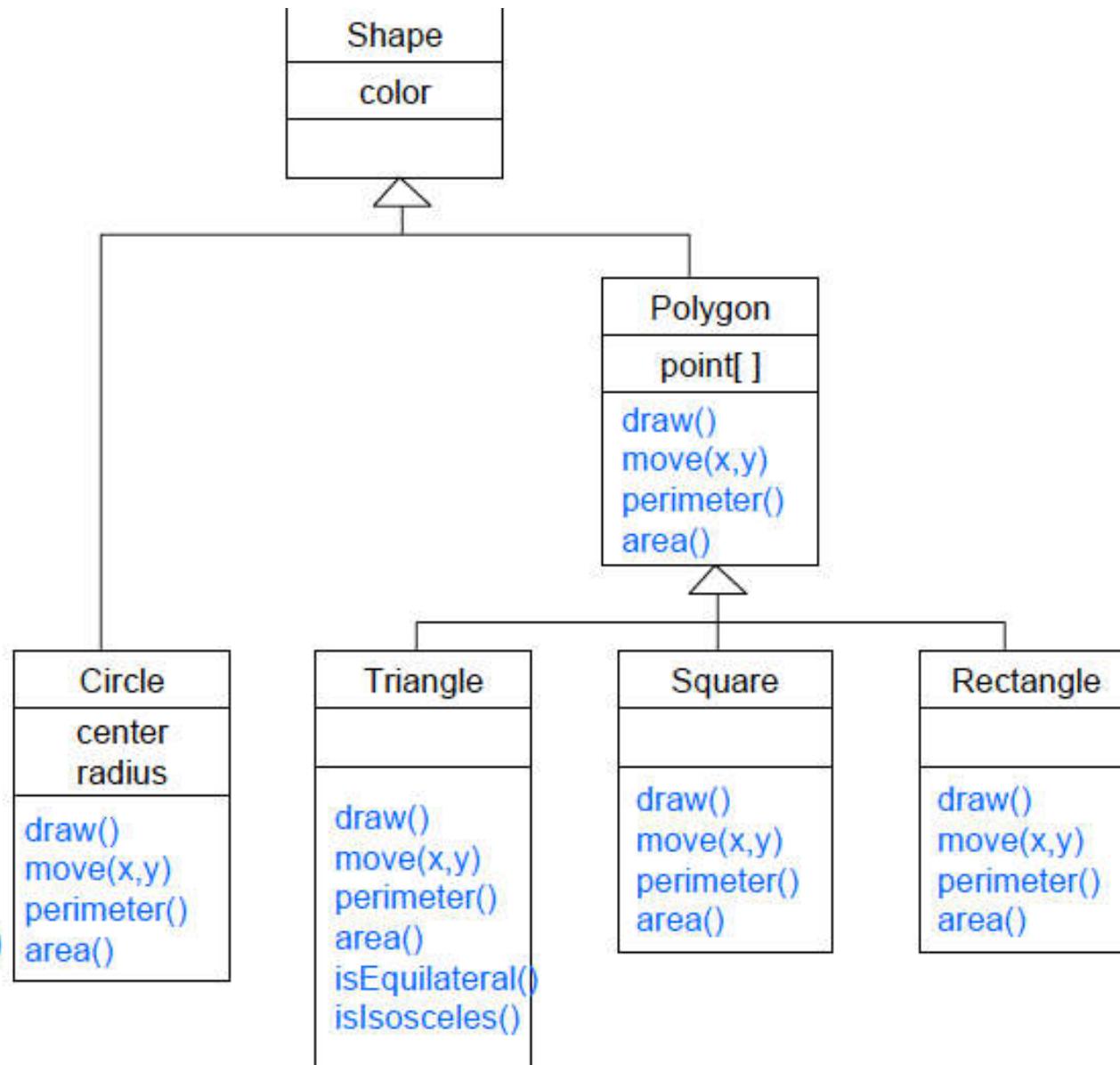
# Methods

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- draw()
- move()
- perimeter()
- area()
- isEquilateral()
- isIsosceles()
- getColor()
- setColor()



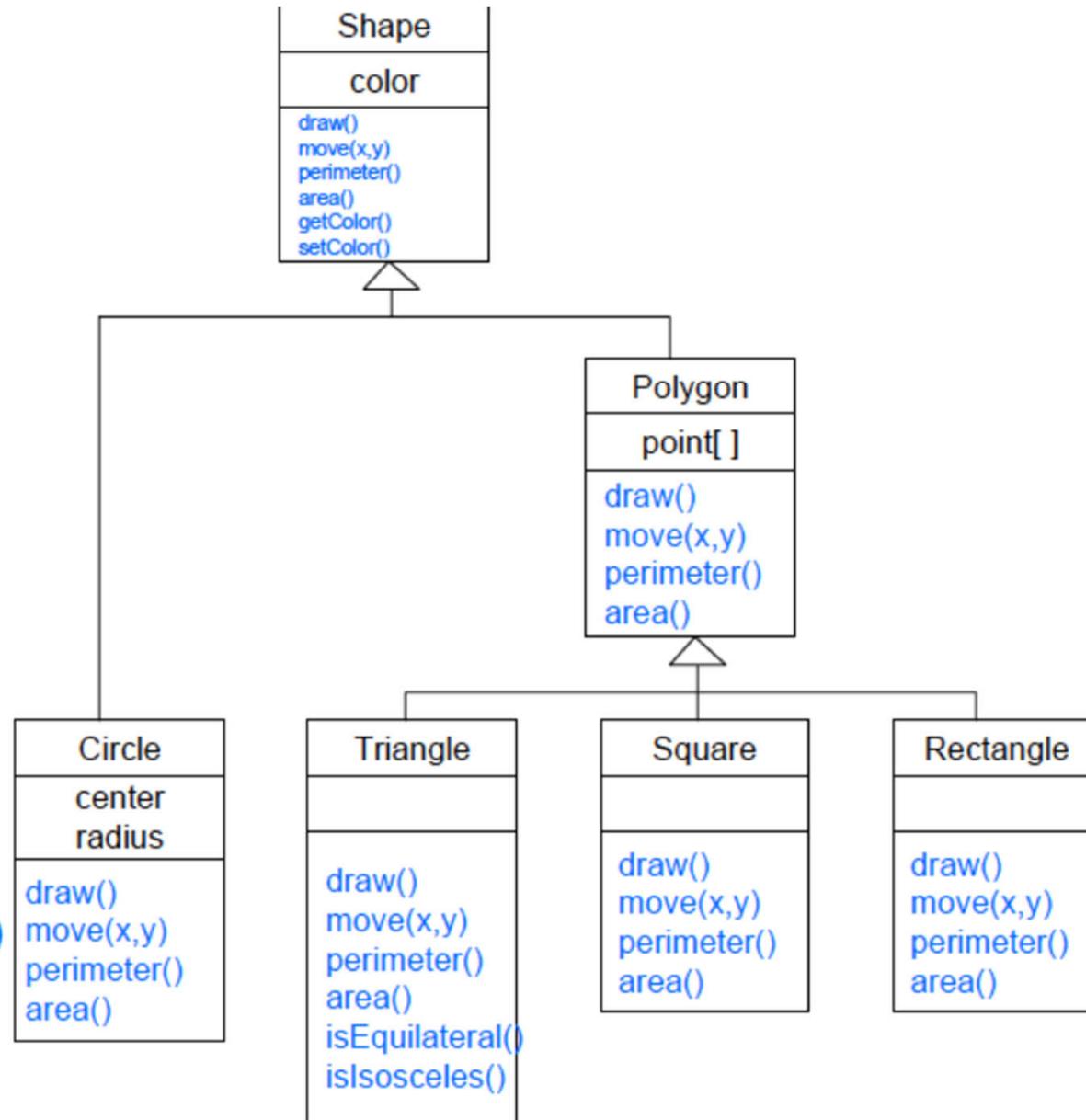
# Methods

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- `draw()`
- `move()`
- `perimeter()`
- `area()`
- `isEquilateral()`
- `isIsosceles()`



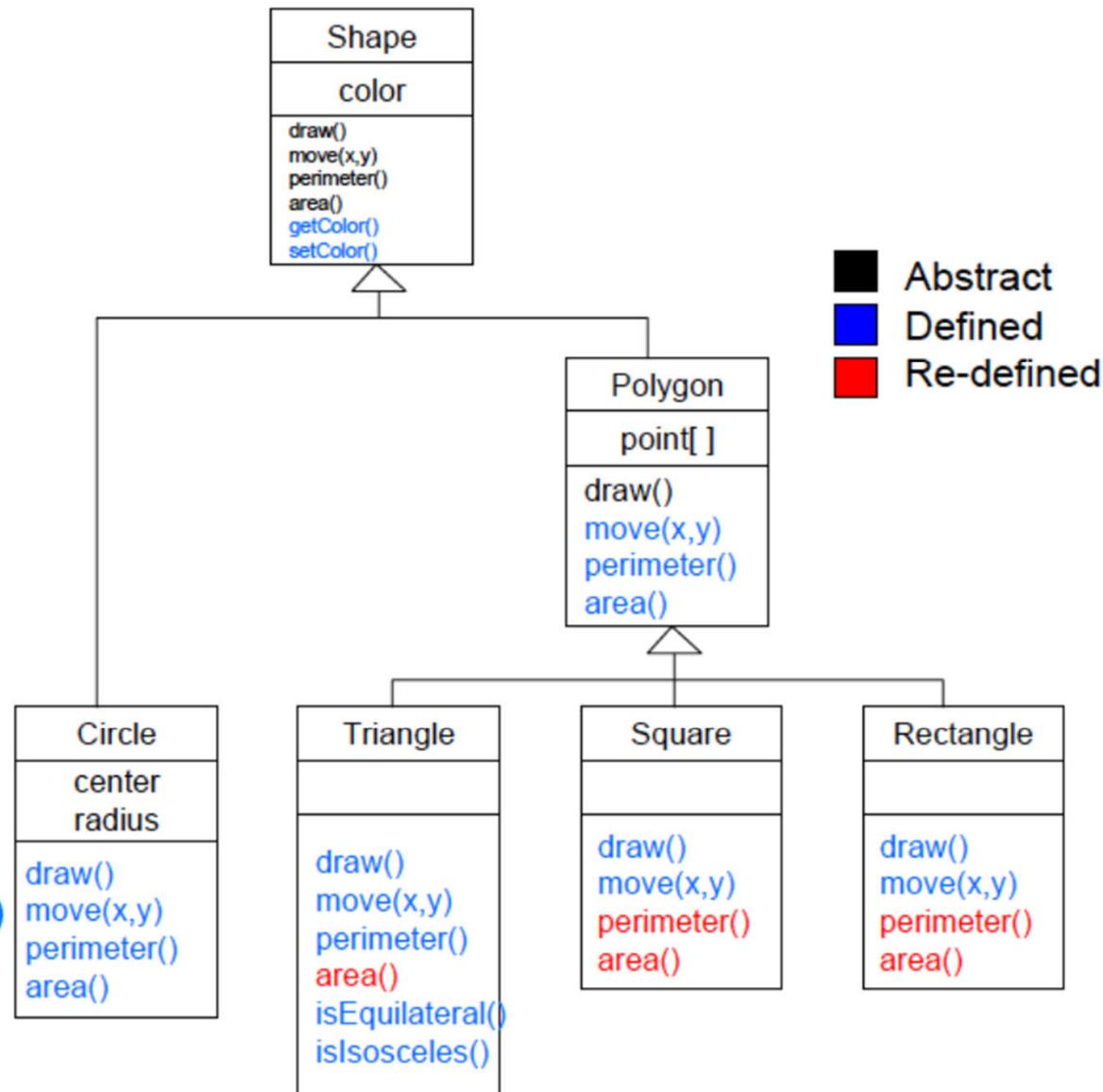
# Methods

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- draw()
- move()
- perimeter()
- area()
- isEquilateral()
- isIsosceles()



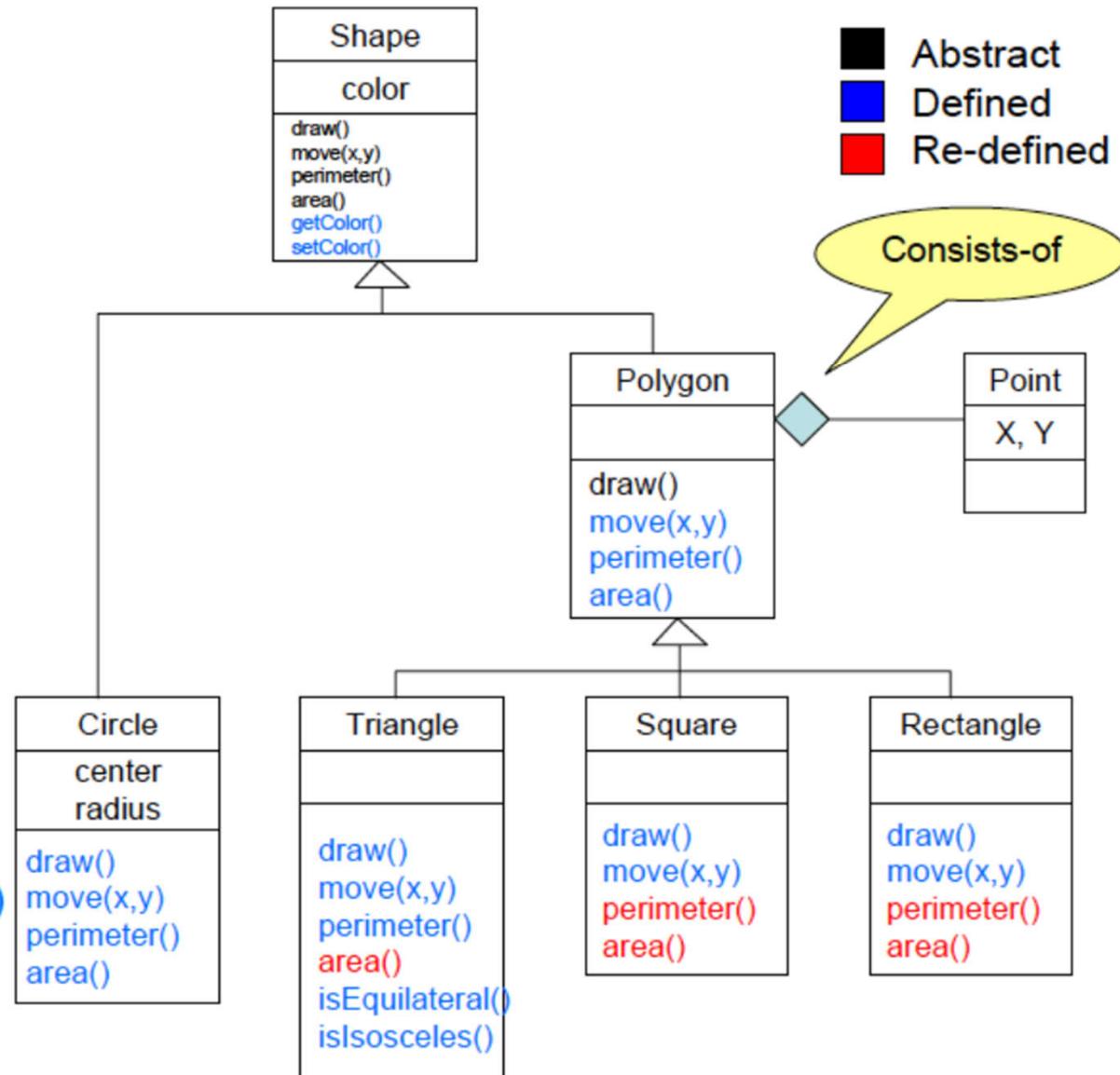
# Methods

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- draw()
- move()
- perimeter()
- area()
- isEquilateral()
- isIsosceles()

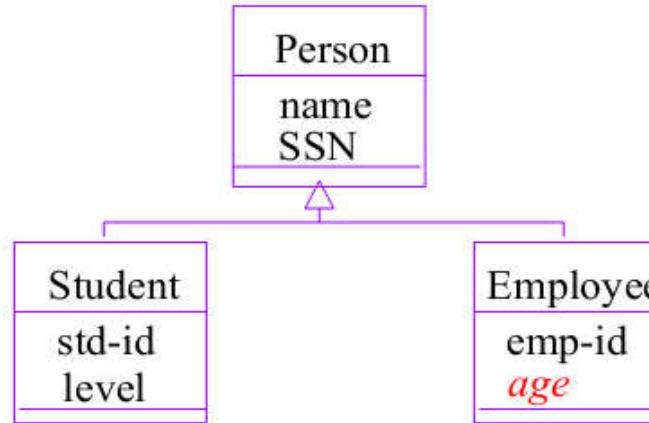


# Final Thoughts

- Shape
- Polygon
- Circle
- Triangle
- Rectangle
- Square
- Point
- Color
- `draw()`
- `move()`
- `perimeter()`
- `area()`
- `isEquilateral()`
- `isIsosceles()`



# Super class vs. Sub class



**Specialization:** The act of defining one class as a refinement of another.

**Subclass:** A class defined in terms of a specialization of a superclass using inheritance.

**Superclass:** A class serving as a base for inheritance in a class hierarchy

**Inheritance:** Automatic duplication of superclass attribute and behavior definitions in subclass.

# Class Diagrams

Class
+ attr1 : int
+ attr2 : string
+ operation1(p : bool) : double
# operation2()

# Class Diagrams

## Attributes

Attributes are shown with at least their name, and can also show their type, initial value and other properties. Attributes can also be displayed with their visibility:

- + public attributes
  
- # protected attributes
  
- private attributes

Class
+ attr1 : int
+ attr2 : string
+ operation1(p : bool) : double
# operation2()

# Class Diagrams

## Operations

Operations (methods) are also displayed with at least their name, and can also show their parameters and return types. Operations can, just as Attributes, display their visibility:

+ *public* operations

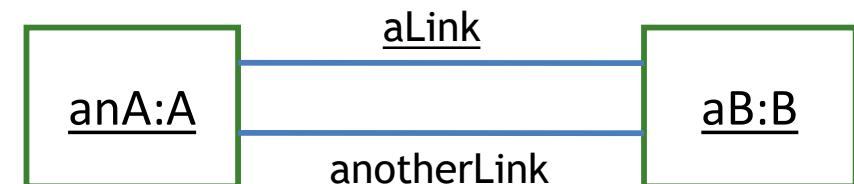
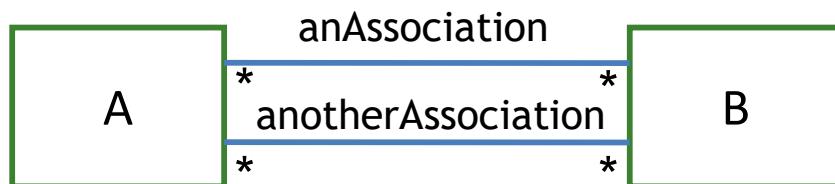
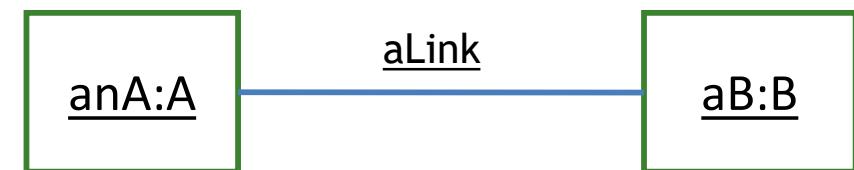
# *protected* operations

- *private* operations

Class
+ attr1 : int
+ attr2 : string
+ operation1(p : bool) : double
# operation2()

# Link & Association

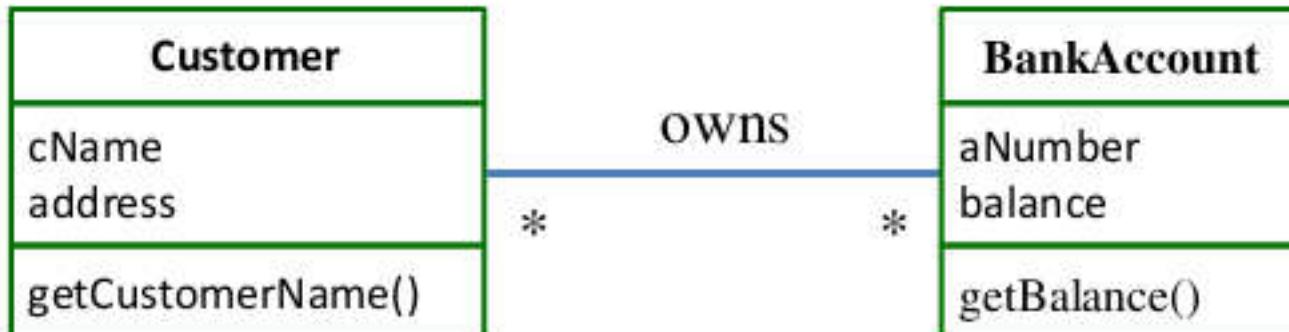
- Link is a physical or conceptual connection among objects
- Association is a description of a group of links with common structure and common semantics



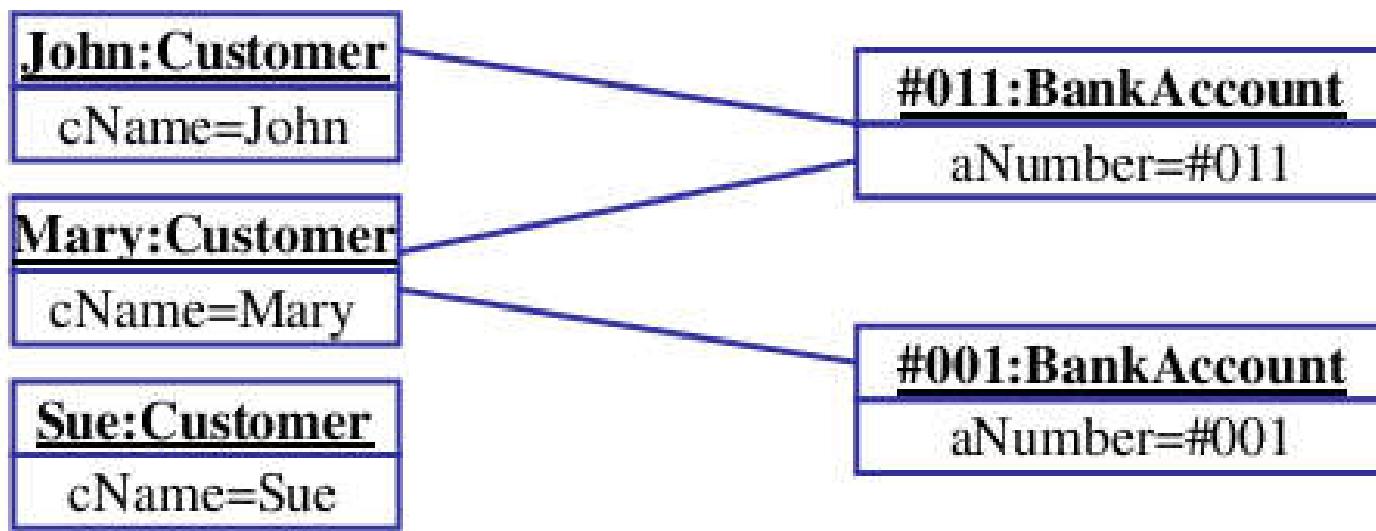
*Class Diagram*

*Object Diagram*

# Association



Class Diagram



Object Diagram

# Multiplicity

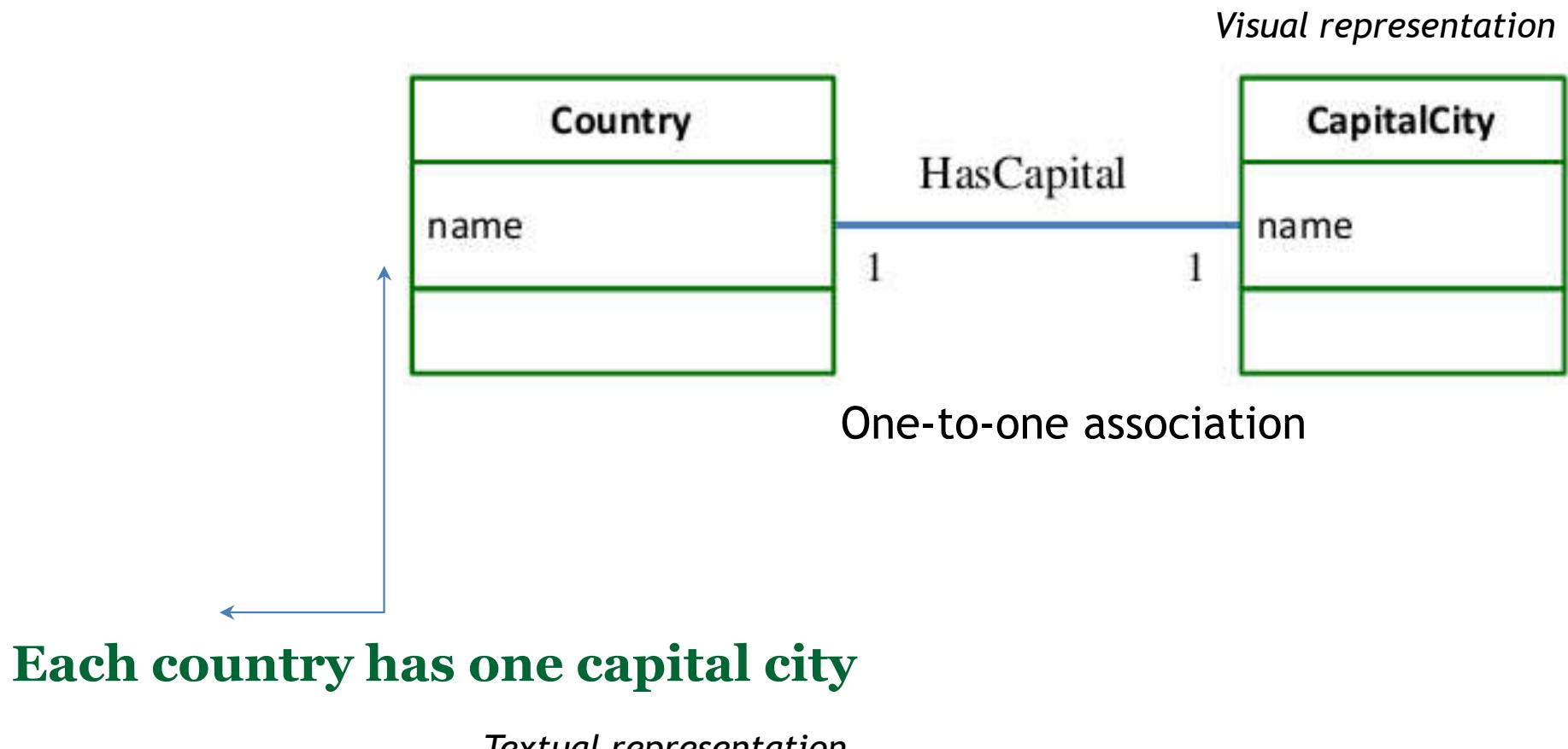
Multiplicity specifies “the number of instances of one class that may relate to a single instance of an associated class”.

- “One” or “Many”
- *Infinite (subset of the nonnegative integers)*

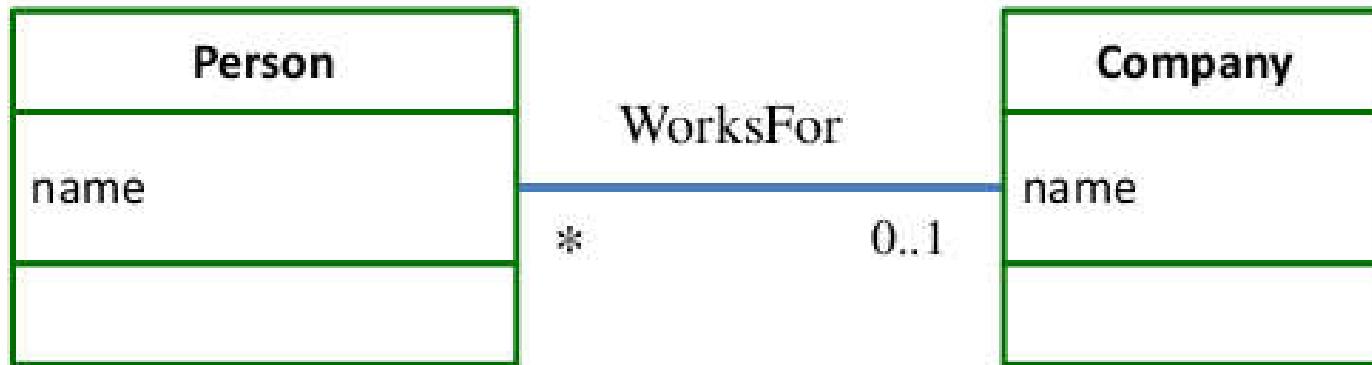
UML Specifies multiplicity with an interval

- “1” (*exactly one*)
- “1..\*” (*one or more*)
- “3..5” (*three to five, inclusive*)
- “\*” (*many*)

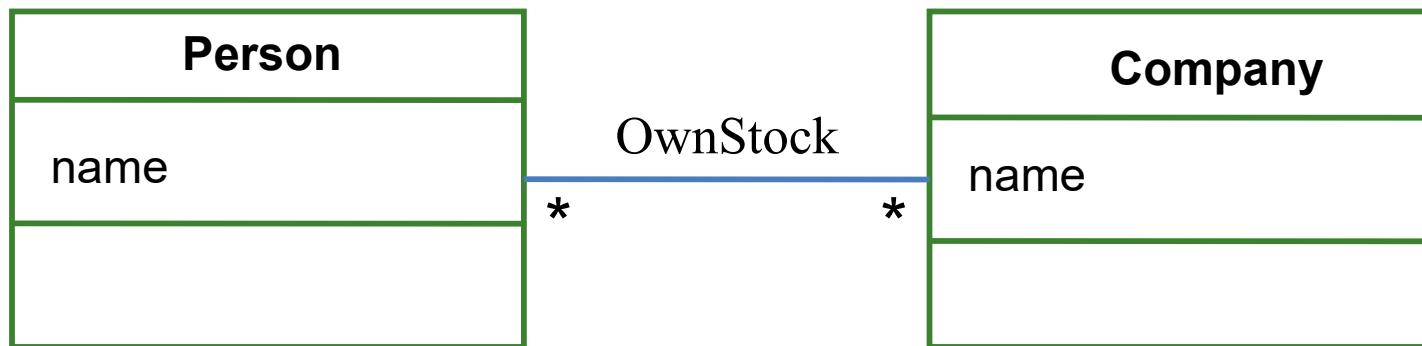
# Association



# Association

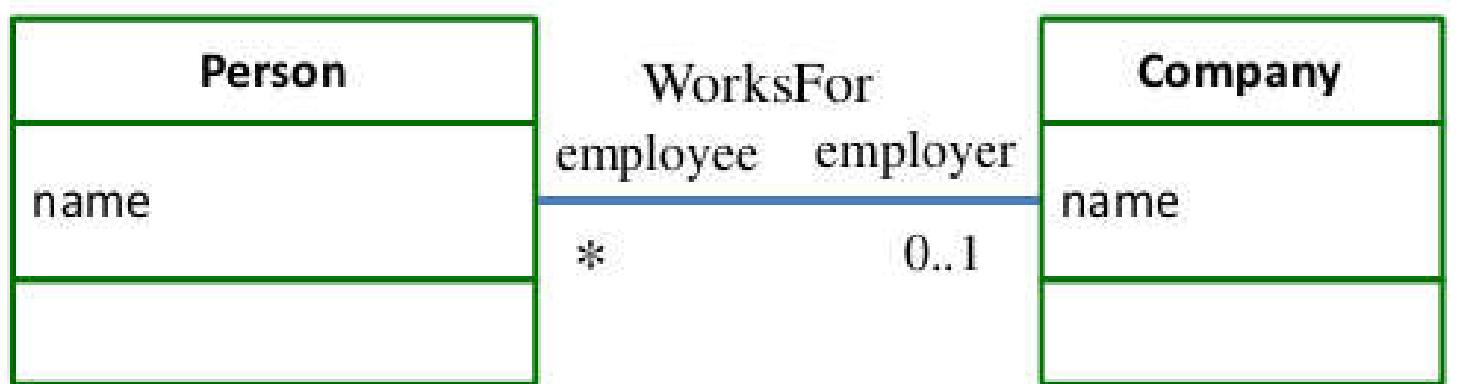


Many-to-one association



Many-to-many association

# Association



*Person and Company participate in association “WorkFor”*

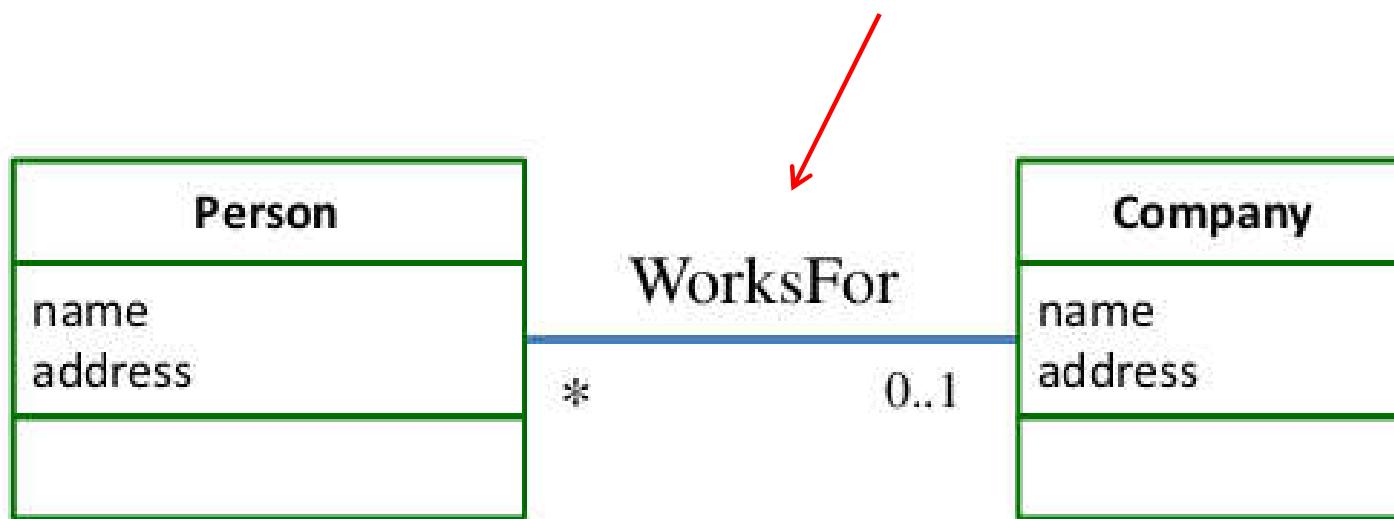
A Person is an employee with respect to a company  
A Company is an employer with respect to a Person

Interpretation?

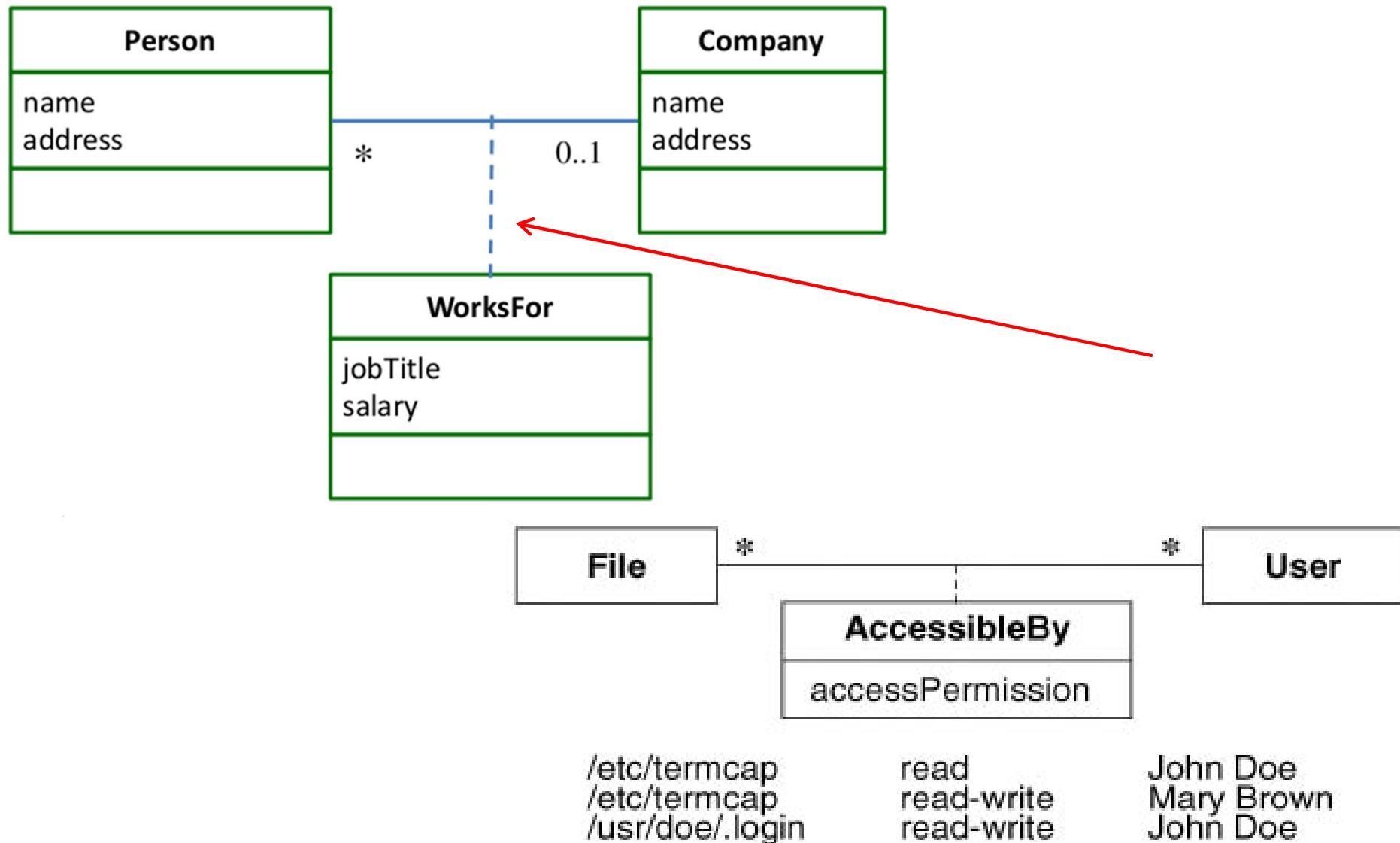
# Association Class

UML offers the ability to describe links of association with attributes like any class.

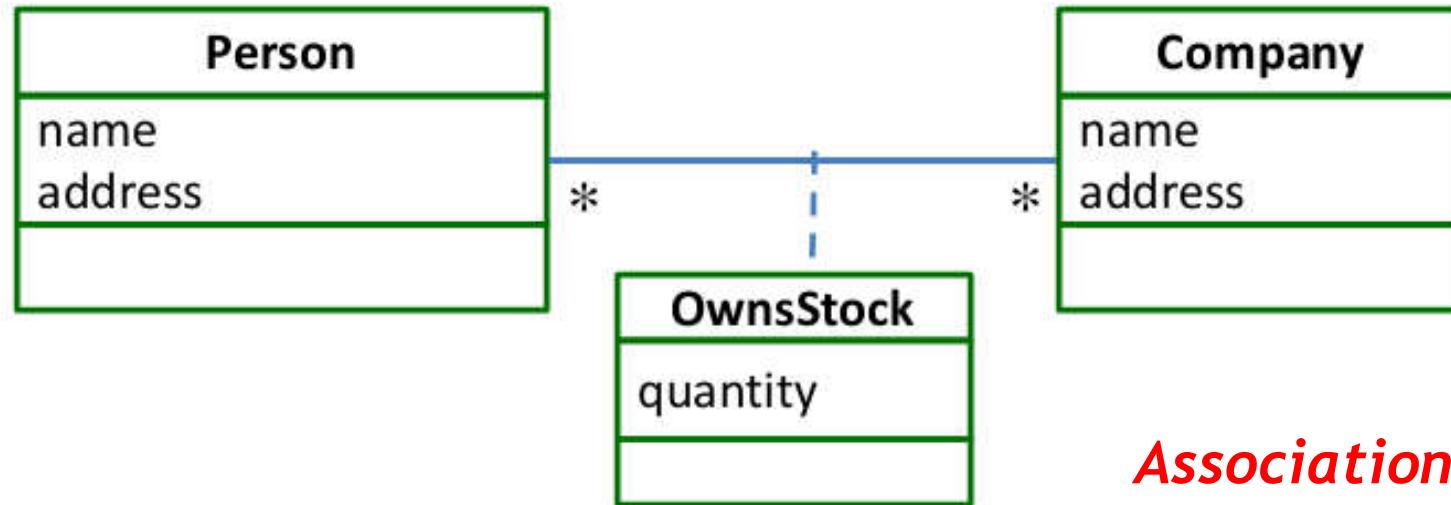
An association class is an association that is also a class.



# Association Class



# Association Class vs Ordinary Class

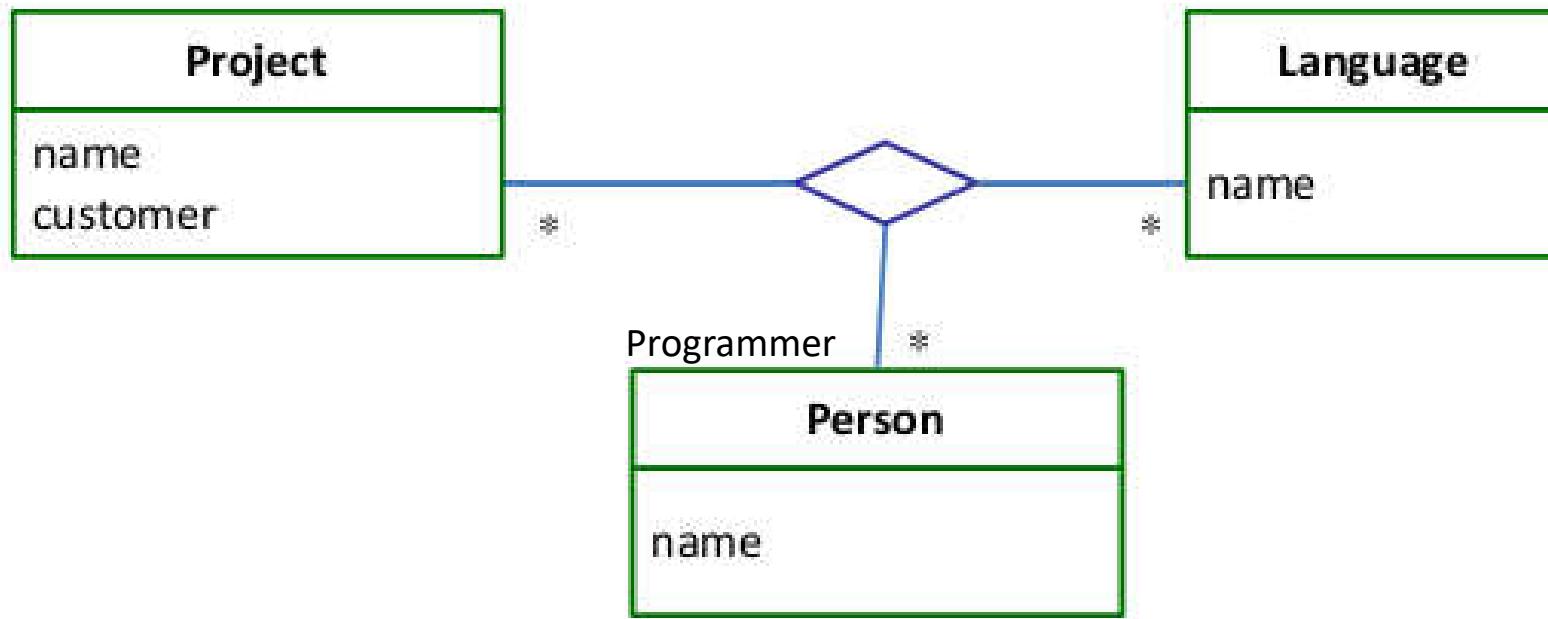


*Association Class*



*Ordinary Class*

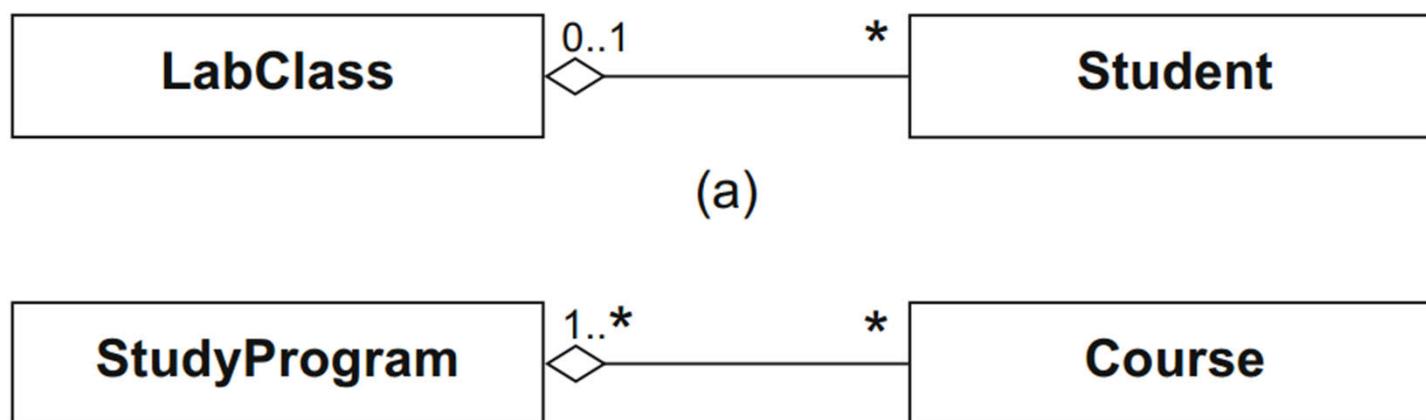
# N-ary Association



*Class Diagram*

# Aggregation

- Aggregation is a special type of Association that shows “**part of**” relationship
- Used to express, instances of one class are parts of an instance of another class
- **Types:** Shared Aggregation and Composition
- Diamond shape is used to represent both

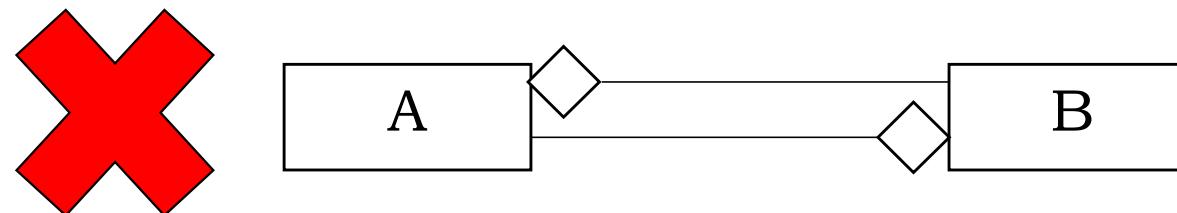
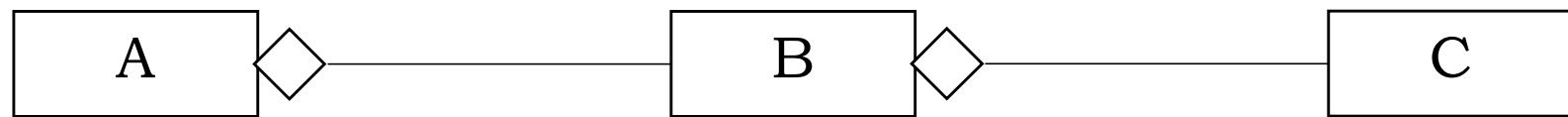


# Shared Aggregation

- Aggregation is used where the “part” entity can exists separately
- “Student” is the “part”
- LabClass is the “whole” in this relation
- The diamond marker is placed at the association end of the class, which stands for the “whole”
- Solid diamond for composition
- Empty diamond for shared aggregation
- **The “part” entity can exist alone/stable.**

# Aggregation

- Shared aggregation and Composition are
  - Transitive
    - if B is part of A and C is part of B, C is also part of A
  - Asymmetric
    - it is not possible for A to be part of B and B to be part of A simultaneously



# Composition



- Composition is used where the “part” entity cannot exist separately
- LectureHall cannot exists without Building
- Beamer may or may not be dependent upon Lecture Hall
- There can be a 1 to 0 relation between Beamer and LectureHall.

# Use-case Relationship

- Two types of Relationship between Use-cases
  - Include
  - Extend
- **Include:**
  - Consists of two uses cases
    - Base use case
    - Inclusion use case
  - “Base” use case includes functionality of “inclusion” use case.

# Use-case Relationship

## ■ **Include:**

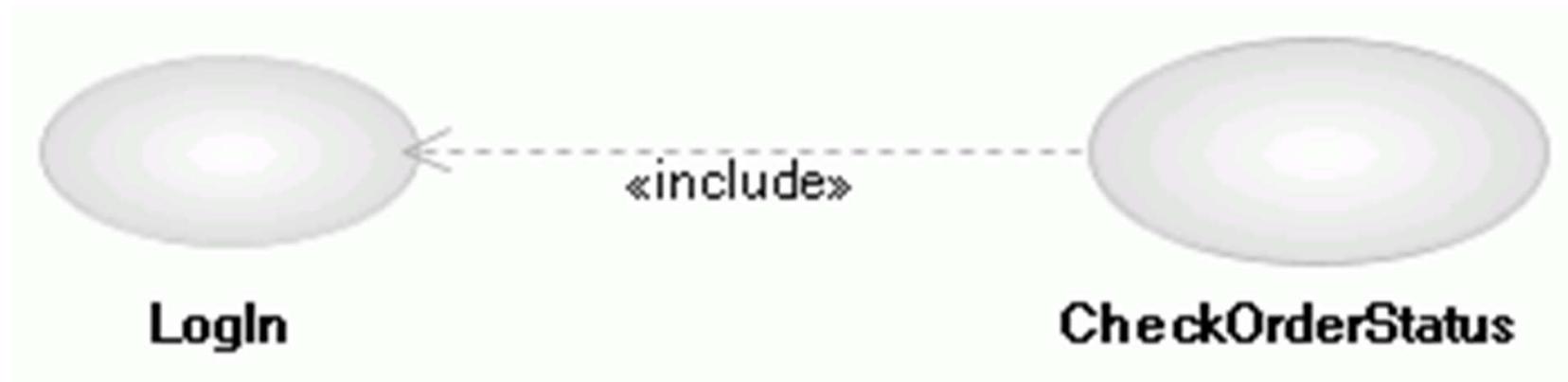
- Indicates re-usability of a task using use-case model
- Behaviour of “inclusion” use case is common for n-number of “base” use cases
- Result or output of the “inclusion” use case is to be used by “base” use case
- Dashed line from “base” to “inclusion”
- Arrow is pointed to “inclusion” use case
- Name of “inclusion” and “base” use cases are generally kept outside the use-case

# Use-case Relationship

## ■ **Include:**

- Indicates re-usability of a task using use-case model
- Behaviour of “inclusion” use case is common for n-number of “base” use cases
- Result or output of the “inclusion” use case is to be used by “base” use case
- Dashed line from “base” to “inclusion”
- Arrow is pointed to “inclusion” use case
- Name of “inclusion” and “base” use cases are generally kept outside the use-case
- “Base” mandatorily includes “inclusion” use case.

# Include Use-case Relationship



The following figure illustrates an e-commerce application that provides customers with the option of checking the status of their orders. This behavior is modeled with a base use case called `CheckOrderStatus` that has an inclusion use case called `LogIn`. The `LogIn` use case is a separate inclusion use case because it contains behaviors that several other use cases in the system use. An include relationship points from the `CheckOrderStatus` use case to the `LogIn` use case to indicate that the `CheckOrderStatus` use case always includes the behaviors in the `LogIn` use case.

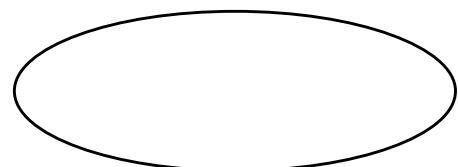
# Extend Use-case Relationship

- One use case extends the behaviour of another use case
- “extension” use case extends the behaviour of the “base” use case
- Relationship indicates “extended” use case is dependent on what happens when the “base” use case executes
- There can be several “extend” use cases for one “base” use case
- Which use case can be “extend” use case
  - A part of a use case that is optional system behavior
  - A subflow is executed only under certain conditions
  - A set of behavior segments that may be inserted in a base use case.

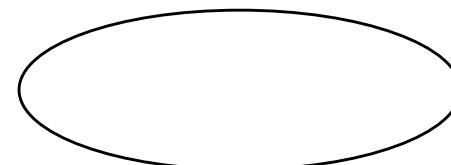
# Extend Use-case Relationship



- Arrow points from “extension” to “base”. Arrow head towards “base”
- Example: You are developing an e-commerce system in which you have a base use case called Place Online Order that has an extending use case called Specify Shipping Instructions. An extend relationship points from the Specify Shipping Instructions use case to the Place Online Order use case to indicate that the behaviors in the Specify Shipping Instructions use case are optional and only occur in certain circumstances.

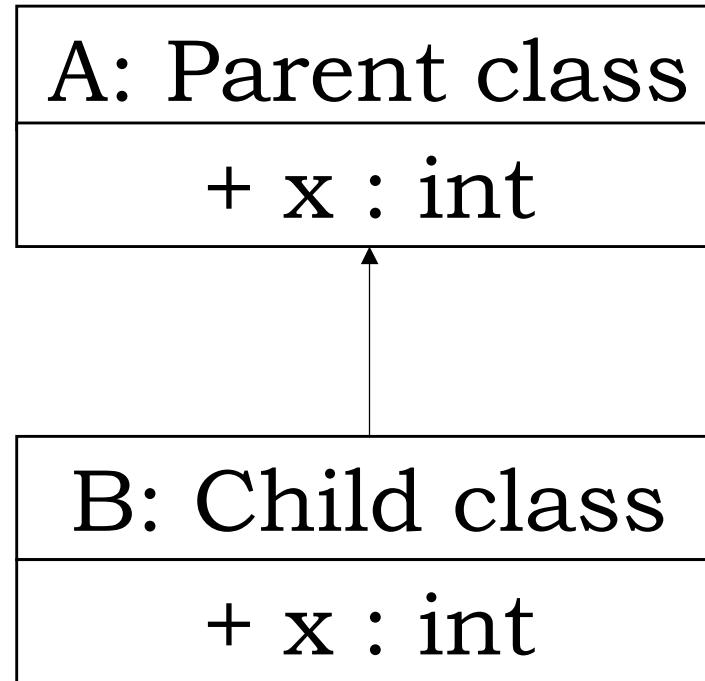


Place Online Order



Specify Gift Wrapping

# Inheritance



- Can we access “x” variables of B and A classes separately?
- How to call members specific to the Base/Parent classes?

How to call members specific to the  
Base classes?

# Super - keyword

- Super keyword is used to refer parent class entities
- Super keyword is used to access entities of parent class from the sub-class
- Super keyword can be used in three scenarios
  - Accessing variables of parent class
  - Accessing methods of parent class
  - Accessing constructors of parent class

# Super – keyword - Rules

- Syntax (from subclass, to access members of parent class)
  - Variables: super.VariableName
  - Methods: super.MethodName( )
  - Constructors: super(<parameter\_list>)
- It also helps to resolve conflicts between same named entities between parent and child class
  - System.out.println(**variable\_data**); //child class variable
  - System.out.println(**super.variable\_data**); //parent class variable
- Calling parent class constructor using super from child class constructor should be the first line.

# Constructor calling - Inheritance

```
class First
{
    First( )
    {
        System.out.println("IIIT");
    }
}

class Second extends First
{
    Second( )
    {
        super();
        System.out.println("Vadodara");
    }
}
```

```
public class
ConstructorCalling_Inheritance
{
    public static void main(String args[])
    {
        Second S = new Second();
    }
}
```

## OUTPUT:

IIIT  
Vadodara

# *this* keyword

- *this* keyword is used to refer to the object which invokes the method
- Whenever a local variable has same name as an instance variable, then the local variable “hides” the instance variable
- *this* keyword helps to use same variable name for both local and instance variables
- Syntax
  - Variables: `this.VariableName`
  - Methods: `this.MethodName( )`
  - Constructors: `this(<parameter_list>)`

# *this* keyword

- *this* keyword should be the first statement within a constructor block
- *this* keyword can be used to call constructor from within another constructor
- *this* keyword helps to reuse the constructor
- Calling a constructor from another constructor is called as explicit constructor invocation
- *this* keyword is used to call constructor of containing class only, where *super* keyword is used to call constructor of superclass.

# *this* keyword

`this.x = x;`

“x” is the instance variable referred by the current object denoted by “this”

Local variable of the block (method or constructor)

- Calling a constructor from a normal method is not possible
- *this* can only call a constructor of same class from another constructor of the same class and not from any normal method.

# Memory Management - Objects

```
class ABC
{
    int x;
    void show( )
    {
        int x = 5;
        this.x = 10;
        System.out.println(x);
    }
}
```

**OUTPUT:** 0 5 0

```
public class TEST
{
    psvm(String[] args)
    {
        ABC a = new ABC( );
        ABC b = new ABC( );
        System.out.println(a.x);
        a.show( );
        System.out.println(b.x);
    }
}
```

# Re-defining variables in Java

- Local variable
- Instance variable
- Class variable
- Local variable:
  - Variables are accessible within the block where they are declared
  - Variables declared within a method/constructor are always accessible from within the method/constructor.
- Instance variable:
  - Variables that are declared within a class
  - Variables which are not within a method/constructor, but inside a class
  - Variables are associated with the object of the class
  - All instances or objects of the class gets separate copy or memory location of the instance variables.

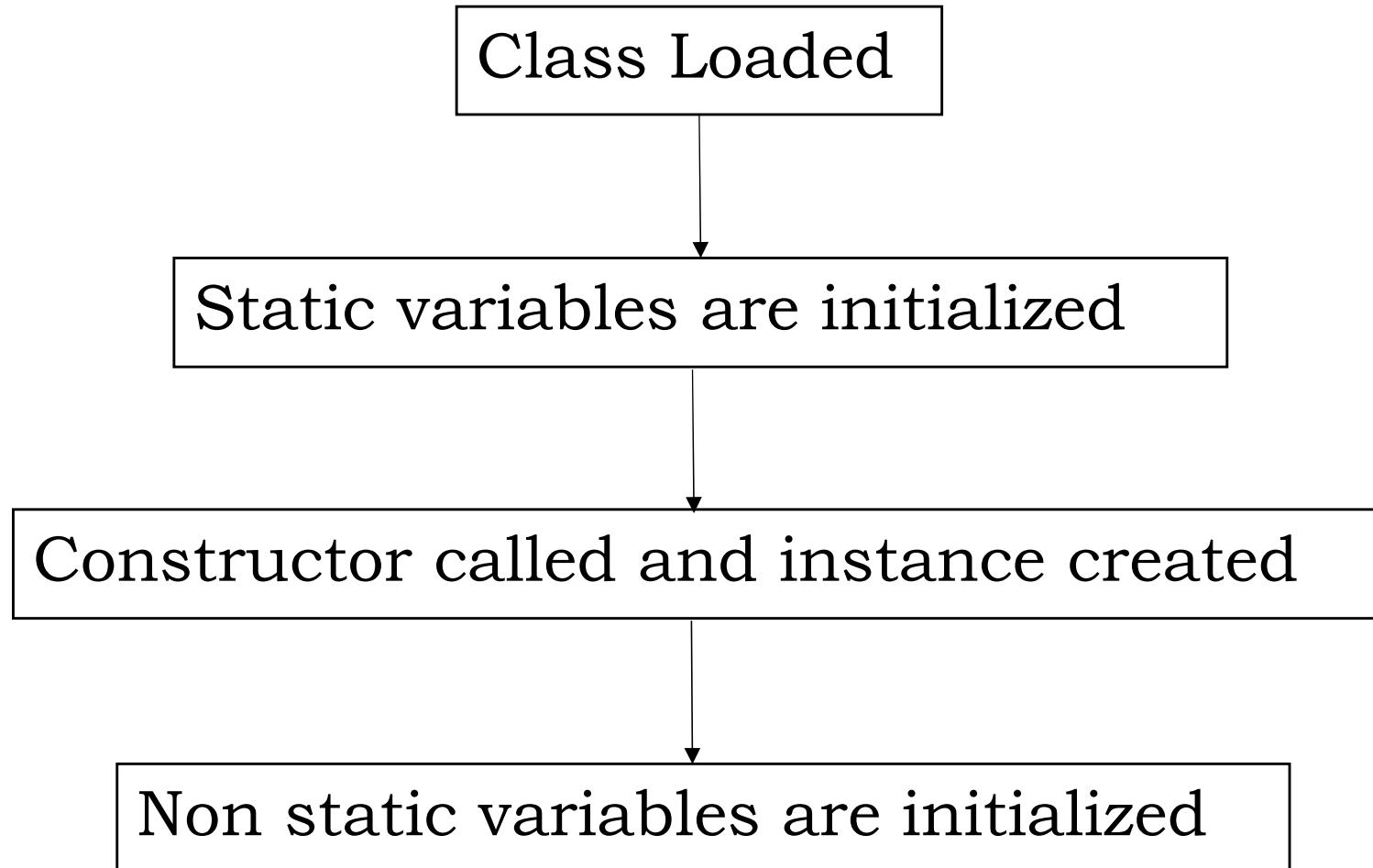
# Re-defining variables in Java

- Instance methods:
  - Methods that are declared within a class
  - Methods are associated with the object of the class
  - All instances or objects of the class gets separate copy or memory location of the instance methods.
  
- Class variable:
  - Variables belongs truly within the class
  - All instances or objects of the class **shares** the same variable
  - Separate memory is not allocated for the class variables with respect to the multiple objects of the class
  - Multiple instances or objects of the class do not get separate copy or memory location of the class variables
  - Class variables are called Static variables
  - Methods that require similar property are called Static methods.

# Static keyword

- Static keyword helps to keep single copy of a variable or method within a class
- Single copy with respect to the multiple objects of the class
- Static variables and methods are Object *independent*
- Therefore static variables and objects can be accessed directly with the name of the class and do not need the class object/instance to access them
- **It is used when a member variable of a class has to be shared between all the instances of the class**
- Static attributes belongs to the class and not to any object of the class.

# Static usage within Java Virtual Machine



# Static keyword - Limitations

- Overriding is not possible on static methods
- Memory is allocated once and never shares different memory
- ‘this’ and ‘super’ keywords cannot be used within a static method.
- **Syntax:**
  - static int x = 10;
  - static int display(int x);

# *final keyword*

- *final keyword* is used to perform the following
  - Makes a variable constant (value cannot be changed)
  - Prevents a method from being overridden
  - Prevents a class from inheriting
- *final variables*
  - final int x = 10;
  - A variable once declared FINAL, can never change its value throughout the program.

# *final* methods

```
class T1
{
    final void display( )
    {
        System.out.println("Pramit");
    }
}

class T2 extends T1
{
    void display( )←
    {
        System.out.println("Mazumdar");
    }
}
```

Error. Final methods  
cannot be overridden

# *final* keyword

- final methods cannot be overridden
- *final* method overloading is always possible
- *final* methods are resolved at compile time
- *final Class*
  - *final* class can contain both final and non-final attributes

# *final* vs *static* keyword

<b><i>static</i></b>	<b><i>final</i></b>
Static methods cannot be overridden.	Final methods cannot be overridden.
Static classes can be inherited.	Final classes can never be inherited.
Static keyword can never make a variable constant. It is shared by all instances of objects and has a single copy.	Final keyword makes a variable constant.
Static variable can change their value during program execution.	Final variable once declared can never be changed.
These are initialized when class loader loads the class.	Final variables are accessible after the object is created.

# Controlling Inheritance

- Make a Class non-inheritable = Final
- Make class members inaccessible = private access specifier
- Allow inheritance + accessibility but work on same copy = static
- Specific instructions in sub-classes = overriding
- Access super class elements from sub-class = super
- Duplicate names with different memory in sub-class = this
- Make a variable un-editable = final.

# Method Overriding - Example

```
class Bank{  
int getRateOfInterest( )  
{ return 0; }  
}
```

```
class SBI extends Bank{  
int getRateOfInterest( )  
{ return 8; }  
}
```

```
class ICICI extends Bank{  
int getRateOfInterest( )  
{ return 7; }  
}
```

```
class AXIS extends Bank{  
int getRateOfInterest( )  
{ return 9; }  
}  
  
public class Method_OVERRIDING{  
psvm(S a[]){  
  
SBI s=new SBI( );  
ICICI i=new ICICI( );  
AXIS a=new AXIS( );  
S.O.P("SBI"+ s.getRateOfInterest( ));  
S.O.P("ICICI"+i.getRateOfInterest());  
S.O.P("AXIS"+a.getRateOfInterest());  
  
}  
}
```

# Abstraction

- An essential element of object-oriented programming is abstraction
- Humans manage complexity through abstraction
- Example; people do not think of a car as a set of tens of thousands of individual parts. They think of it as a well defined object with its own unique behavior
- This abstraction allows people to use a car without being overwhelmed by the complexity of the parts that form the car. They can ignore the details of how the engine, transmission, and braking systems work
- Instead they are free to utilize the objects as a whole.

# Abstraction

- Process of representing essential features without including the background details or explanations
- Only the structure of the class or a method is declared
- **It does not contain any body or statement within it, i.e. they are not defined within the class considered as abstract**
- Structure or statements mentioning the tasks to be performed by an abstract method is defined inside the corresponding subclass
- The superclass or the class considered as abstract contains only the method declaration without any definition of how to perform a task
- The subclass after inheritance, defines (assigns) tasks to the abstract method in the superclass.

# Abstract in Java

```
abstract class test5
{
    void show( ) {
        System.out.println("Mazumdar");
    }

    abstract void display( );
}
```

```
class check1 extends test5
{
    void display( ) {
        System.out.println("Pramit");
    }
}
```

```
public class check
{
    public static void main(String args[])
    {
        check1 C = new check1( );
        C.display( );
        C.show( );
    }
}
```

We try not to define a  
normal methods  
inside an abstract  
class

# Abstract in Java

- Classes considered as abstract cannot have Objects
- Used in scenarios where there is no need to create object of a class throughout the development cycle
- They are always a superclass in an inheritance, thus often known as the abstract superclass
- Abstract class showcases a superclass from which other classes can inherit and thus share the common design methodology
- The subclasses that inherit an abstract superclass are known as the concrete classes, as they are actually providing the definition to the abstract methods
- Abstract superclass is too general to create an object. They only specify what is common among the subclasses.

# Abstract in Java

- Syntax:

```
abstract class test
```

```
{
```

```
.....
```

```
}
```

- Abstract class contains multiple abstract methods
- It may also contain normal methods, constructors, and variables
- Abstract methods within an abstract class do not have any definition
- Abstract methods are defined within the subclass after inheritance.

# Abstract properties

- Abstract methods cannot be static
- Abstract class can have instance methods/variables, static methods/variables, final methods/variables, and constructors
- Abstract classes cannot have objects
- Abstract methods of a class MUST BE defined in the subclass. You should not leave a method declared but not defined !!
- Constructors cannot be abstract
- Normal classes cannot contain abstract methods. Abstract methods can only be within Abstract Classes.

# Abstract limitation

- Abstract keyword may be used to perform abstraction in Java
- An abstract superclass is extended by a normal subclass
- Abstract methods in the abstract superclass is defined within the normal subclass
- According to Java properties, multiple inheritance is not directly possible
- Therefore, **a normal subclass can extend or define methods of a single abstract class**
- **Normal subclass cannot extend multiple abstract superclasses.**

# Interface

- **Multiple inheritance can be performed in Java using an Interface**
- Interfaces are syntactically similar to classes
- Interfaces do not have objects
- Interfaces do not have instance variables
- Methods inside Interfaces are declared and not defined
- A single class can ‘implement’ multiple Interfaces
- Public and Default access specifier is possible within an Interface
- Private access specifier is not possible within an Interface.

# Interface

- Interface specifies what a class must do, but not how it does it
- A class can ‘extend’ a single class
  - `class test1 extends test2 { ..... }`
- A class can ‘implement’ multiple interfaces
  - `class test1 implements test2, test3{ ..... }`
- An interface can ‘extend’ multiple interfaces
  - `class test1 extends test4, test5{ ..... }`
- A class can ‘extend’ a single class AND ‘implement’ multiple interfaces
  - `Class test1 extends test2 implements test4, test5{ ..... }`

# Interface

- Interface contains;
  - Variables which are static
  - Variables which are final
  - Methods which are Abstract
- Abstract classes can ‘implement’ Interfaces
- All abstract methods declared in the Interface must be defined in the implementing class
- ‘implements’ and ‘extends’ both depicts ‘is-a’ relationship.

# Package

- Packages are used to group a variety of classes and/or interfaces together
- Grouping is done by programmer according to the functionality
- Packages provides a way of separating coding from design
- Classes contained in packages can easily be reused
- It provides a way to ‘hide’ classes thus preventing other programs or packages from accessing classes
- A class defined within a package must be set with an access specifier by which other package classes can access.

# Package creation

- Create a folder with same name as the desired Package name
- Create a normal java file and write the class definition within it
- First line should have,
  - package <package name>;
- Access specifiers provided for the class should agree with the requirements
- Place the java file in the same folder with the package name
- This enables to create multiple classes within the same package
- Typically classes with similar properties/objectives are kept within the same package.

# Package creation

- Java files present in a package need not be compiled
- Classes within a package are not executed directly
- Application programs are written outside package to use the entities present within the class
- Create a new java file. Outside the package folder. Within the same path
- This would be the application program which would be written to access the functionalities/methods defined/declared within classes present in a package
- Application program (java file) that uses a class (java file) present within a package must IMPORT it.

# Package creation

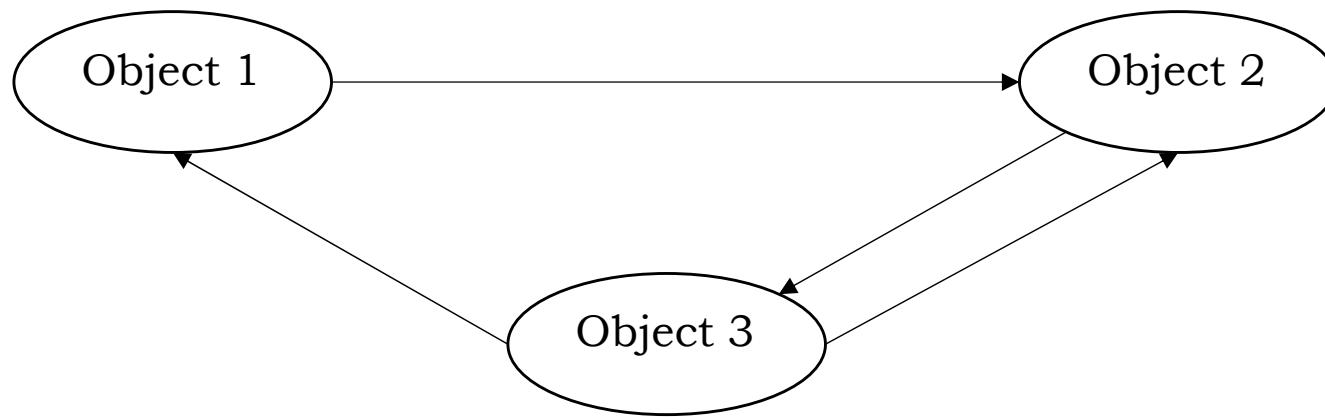
- Syntax;
  - `import <package name> . <classname> ; //one class within the package folder`
  - `import <package name> . * ; //all classes within the package folder`
- The application program is compiled and executed just like a normal java file
- Separate access specifiers for class and methods within the classes
- **If class is public and a method is private, then that method would not be accessible to other classes**
- **Thus every method to be accessed by outside classes need to be assigned the correct access specifier.**

# Elements of Object Models

- Four major elements of the Object Model
  - *Abstraction*
  - *Encapsulation*
  - Modularity
  - Hierarchy
- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - **Concurrency**
  - Persistence

# Concurrency

- Concurrency is the property that distinguishes an active object from one that is not active



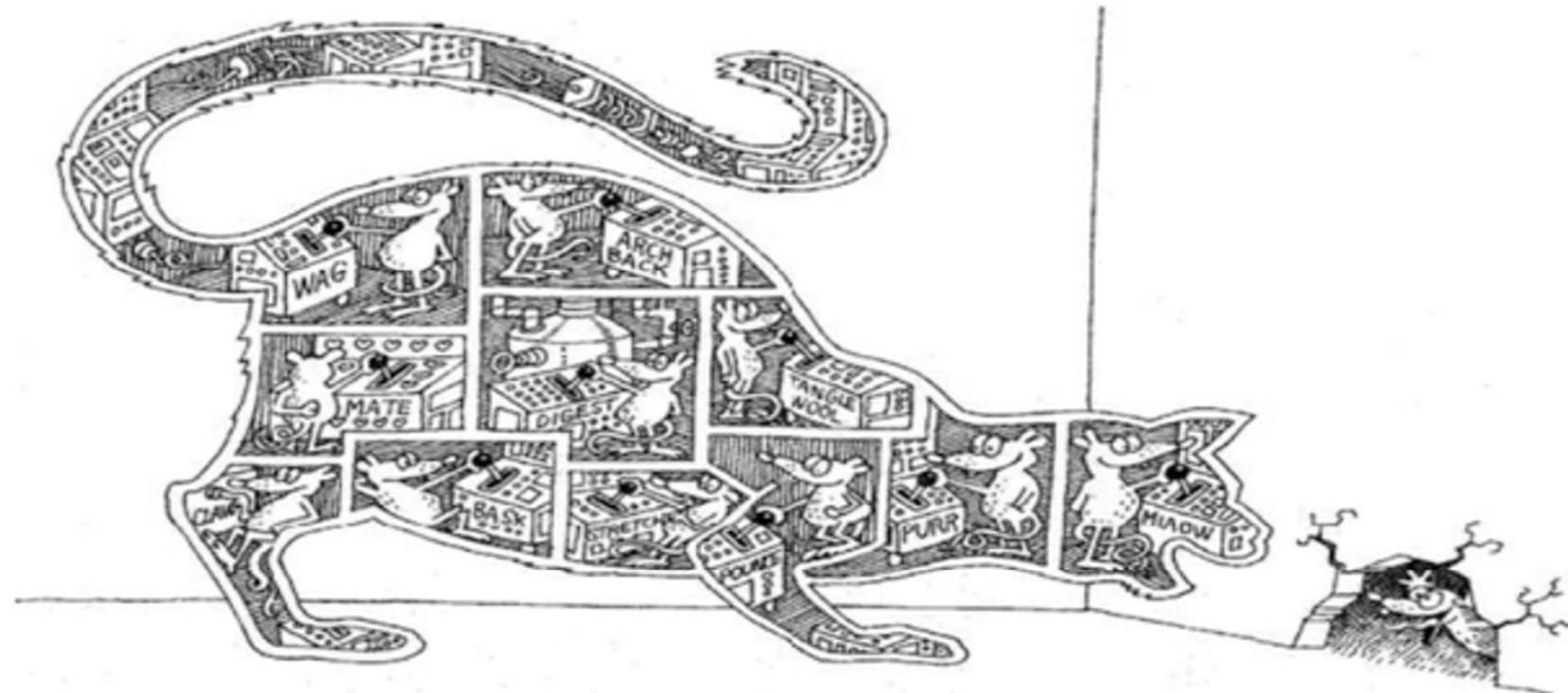
- Should we design a system which states how or in which sequence the messages should flow?
- We need to design a system in which at any point of time ... any object can send any message and can operate on any data ... without any restrictions .... Yet the system must run smooth !!

# Concurrency

- Possible scenarios:
  - Object 1 is sending a message to Object 2 requesting for a service
  - Object 1 is receiving a message from Object 3 to provide a service
  - System should be designed and implemented in such a way that both are possible at a give time instance
  - Similarly at the same instance there might be other objects in the system who are in an idle state doing nothing
- A Client-Server model can never function without enabling Concurrency !!
- We should never create a situation where we define which object sends what message at what timestamp, such scenario would take the fun of using a system !!

# Concurrency

- Allows multiple tasks to execute, interact, and collaborate at the same time to achieve the global functionality



*Concurrency allows different objects to act at the same time*

Source: *Object-Oriented Analysis and Design – With Applications* by Grady Booch et. al. (3rd Ed, 2007)

# Concurrency: Heavyweight and Lightweight

## ■ Heavyweight Concurrency:

- Typically independently managed by the target OS and has its own address space
- Communication among heavyweight processes is expensive and involves inter-process communication
- Commonly called as Process

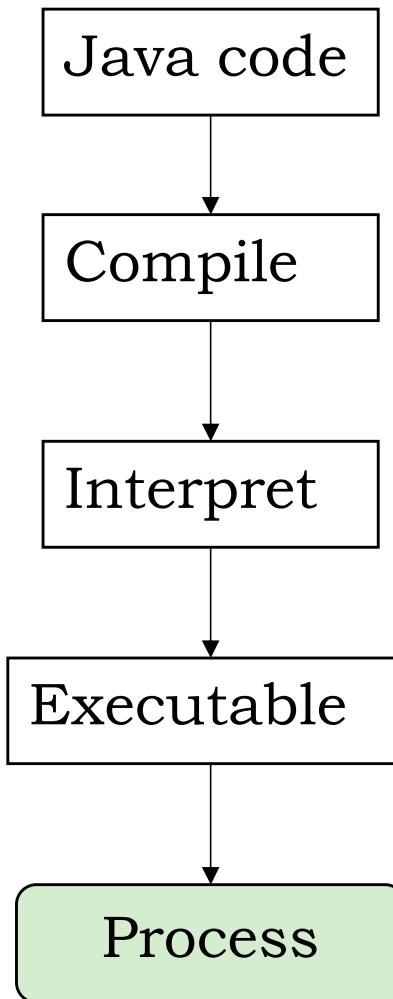
## ■ Lightweight Concurrency:

- It lives within a single OS process along with other lightweight processes and share the same address space
- Communication among lightweight processes is less expensive and often involves shared data
- Commonly called Thread.

# Process

```
class T1
{
    public static void main(String args[])
    {
        int sum = 0;

        for( int i=0; i < 10000000; i++ )
        {
            sum = sum + i;
        }
    }
}
```



# Process

- Problem: Loop would run from 0 to 10000000 (10 million) times
- Addition needs to be done at each step
- 10 million number of times the same set of instructions (tasks) need to be performed
- The numbers after 5 million need to wait for execution until the numbers before them are executed.

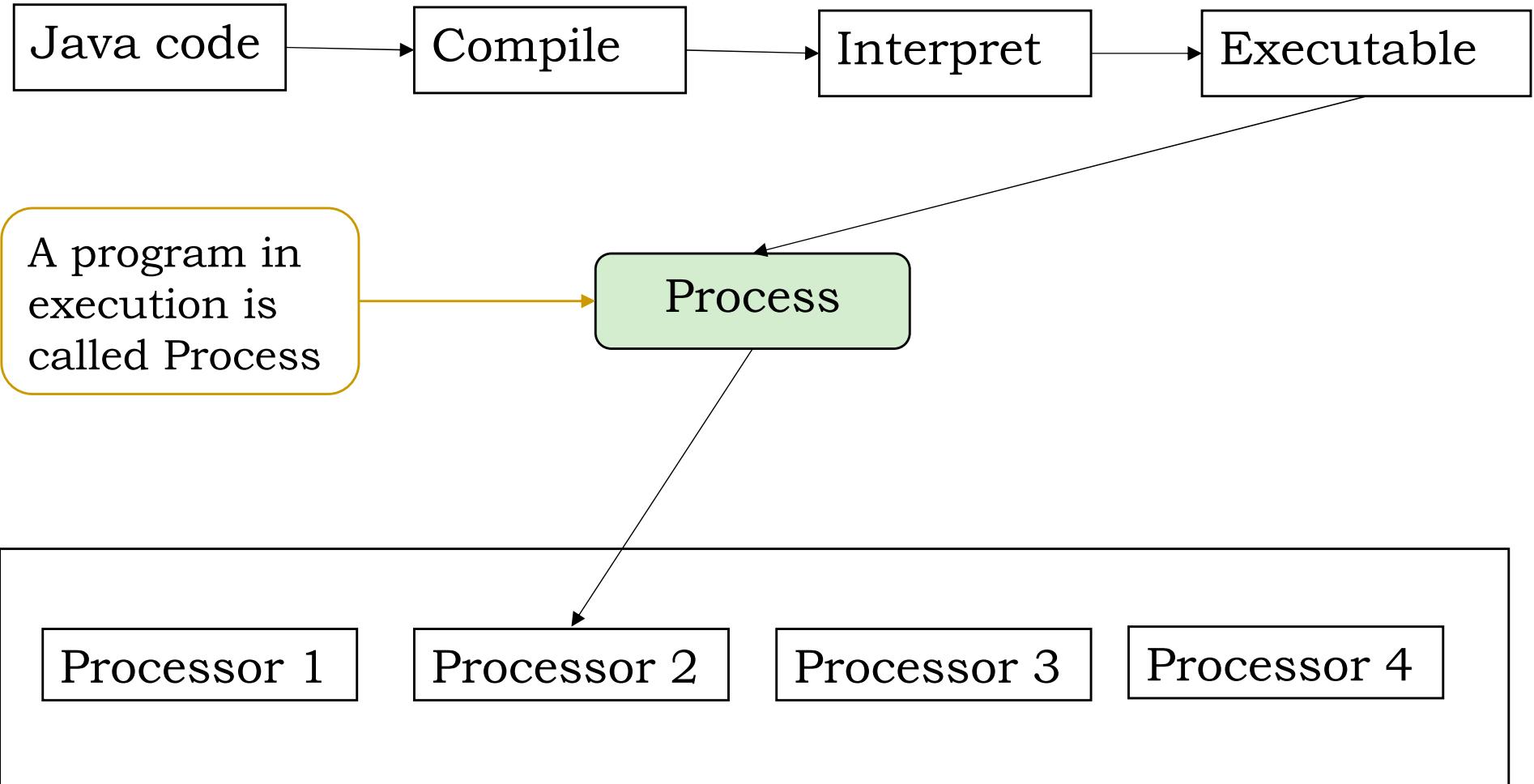
Processor 1

Processor 2

Processor 3

Processor 4

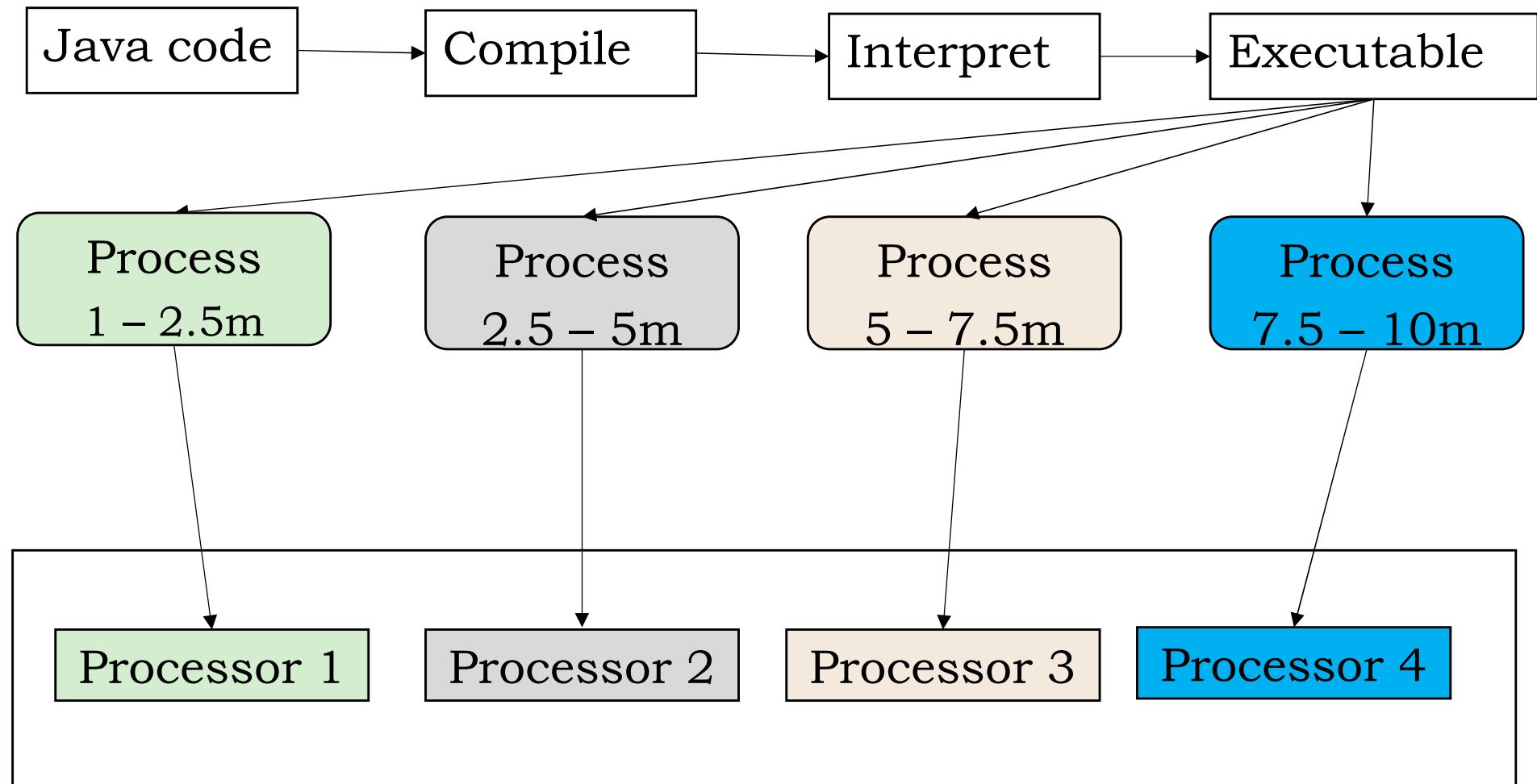
# Process



# Process

- Only one processor is used
- Other processors are left unused
- Time consuming
- Optimized execution would be to utilize all the processors present in the system
- Solution:
  - Divide the task into parts
  - 10 million can be divided into 4 parts (2.5 million each)
  - Each task can be assigned separate process
  - 4 processes each with 2.5 million numbers
  - Each process assigned to separate processors
  - 4 process executes on 4 processors

# Process



# Process

- What we achieved:
  - Parallelization is achieved
  - Speed up the execution
- The processes are isolated from each other
- Thus each process has its own;
  - Set of instructions
  - Data
  - Heap
  - Stack, etc.
- Question: Finally the results of each process needs to be combined .... Who does that .. Who monitors it??

# Process

- Interprocess communication (IPC) which communicates among the processes in execution
- Either they keep track of the execution timeline of each process running in parallel
- Or they provide a separate shared memory where results of each parallel process is stored, and eventually used for computing the final result
- This mechanism is extremely expensive
- It is desired not to rely on such mechanism if alternative is available.

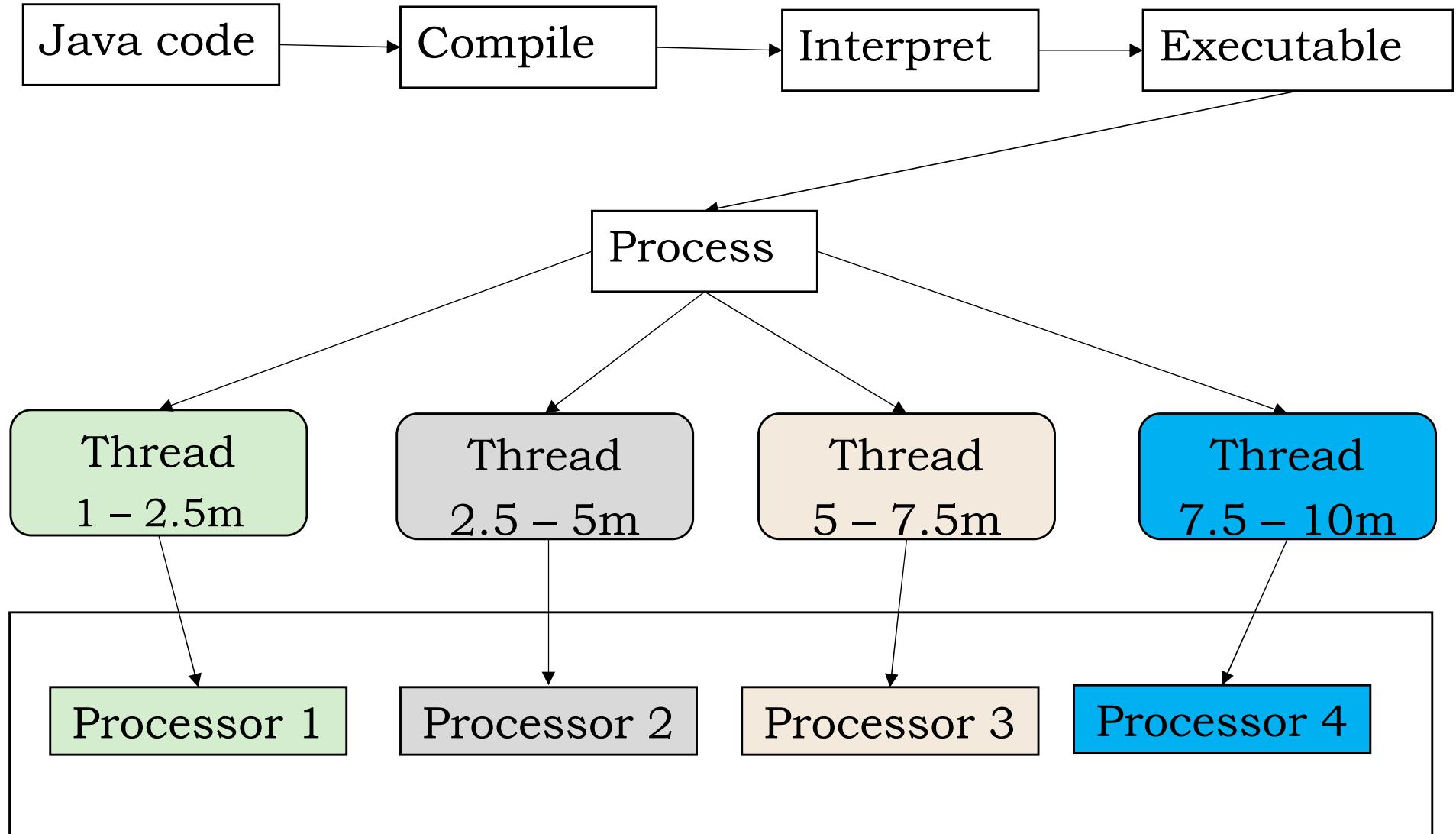
# Process

- Why creation of 4 separate process is a problem?
- All process execute same set of instruction (adding values)
- All process access the same array (from a different location)
- So instructions and data are duplicated in all the 4 parallel processes
- THREADS can help in addressing this problem.

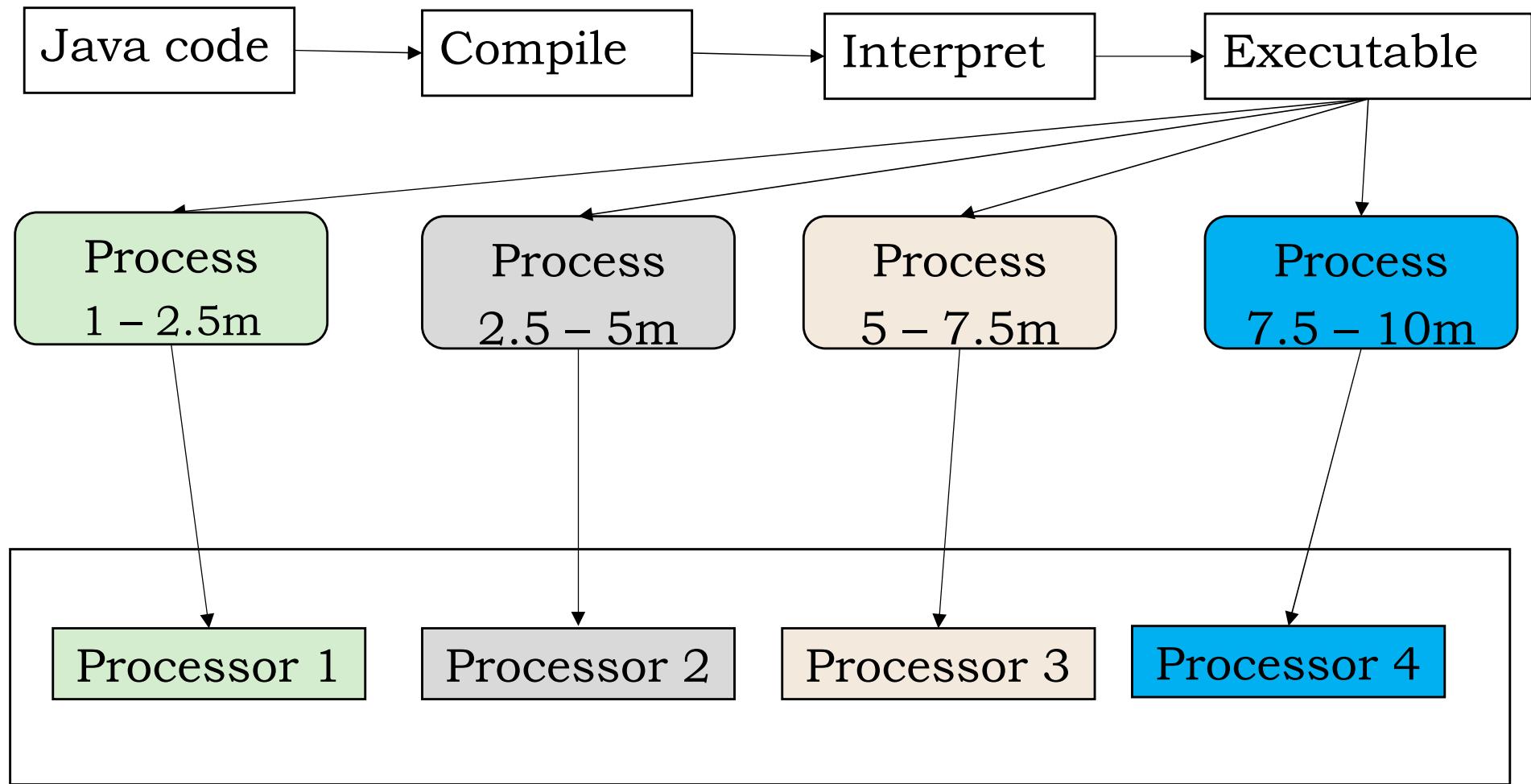
# Thread

- Threads are extremely light weight when compared to processes
- They require less resources than a complete process
- Instead of creating 4 parallel processes. We may consider creation of 4 parallel threads
- Assign 4 threads to 4 processors present in the system
- One process => 4 threads => 4 processors
- Each thread with 2.5million numbers.

# Thread



# Process



# Thread

<b>Thread</b>	<b>Process</b>
Thread has no data segment or heap to store/access data	Process has code, heap, stack, and other segments as well
Thread cannot live on its own. It is always associated with a process	A process can execute independently.
There can be one or more threads associated with a process	Every process has at least one thread for execution
Threads within a process share the same code, resources, etc.	Separate process have separate code, memory, resources, etc.
Thread has its own Stack. On thread destroy the stack is reclaimed by the host process.	If a process dies, all its threads are also destroyed.

# Thread

- A sequential program has only one single flow of control
- Thus at any one timestamp there is only one instruction that is being executed
- A program written sequentially also consists of only one thread that initiates the execution process
- A multithreaded program has multiple flows of control when executed

# Thread

- **A program is divided into a number of parts**
- Each part defines a specific task
- **Each task/part of the program is called as the threads of the program**
- Multithreaded program is one which consists of multiple threads within it and all threads run in parallel, known as Multithreading.

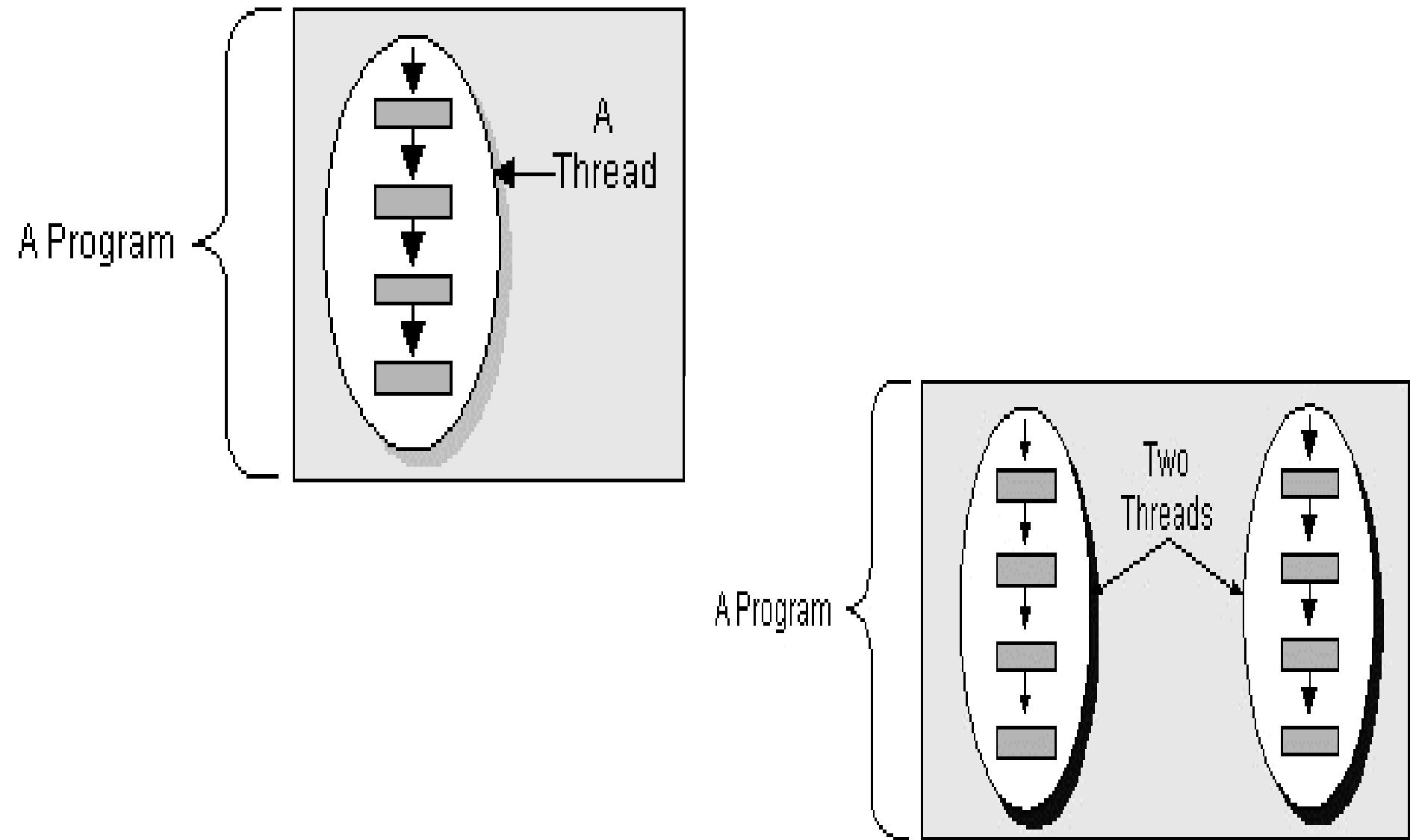
# Thread

- Process is an isolated execution entity which has its own code, heap, stack, and other segments
- Thread does not have its own data segment and heap
- Thread uses the data segment and heap of host process
- **A thread within a process will use the same resources as allocated to the host process**
- **Thus process in itself is a complete entity and thread is a part of it.**

# Thread

- **A process cannot execute without a thread**
- **“main” is one thread that always executes in a program**
- Threads in java can be considered as sub programs of the main application program which shares the same memory space
- Threads in Java can be executed in parallel thus boosting parallelism through Java.

# Thread



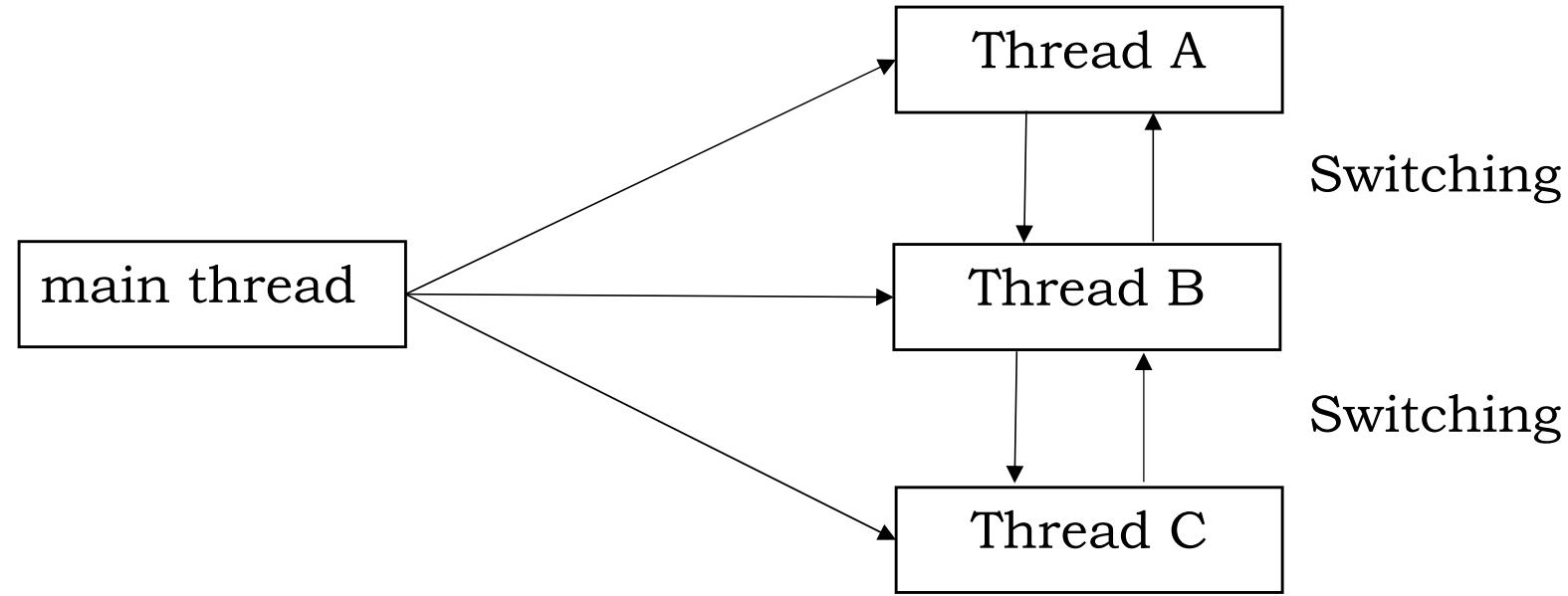
# Thread

```
class ABC
{
    public static void main(String x[])
    {
        .....
        .....
    }
}
```



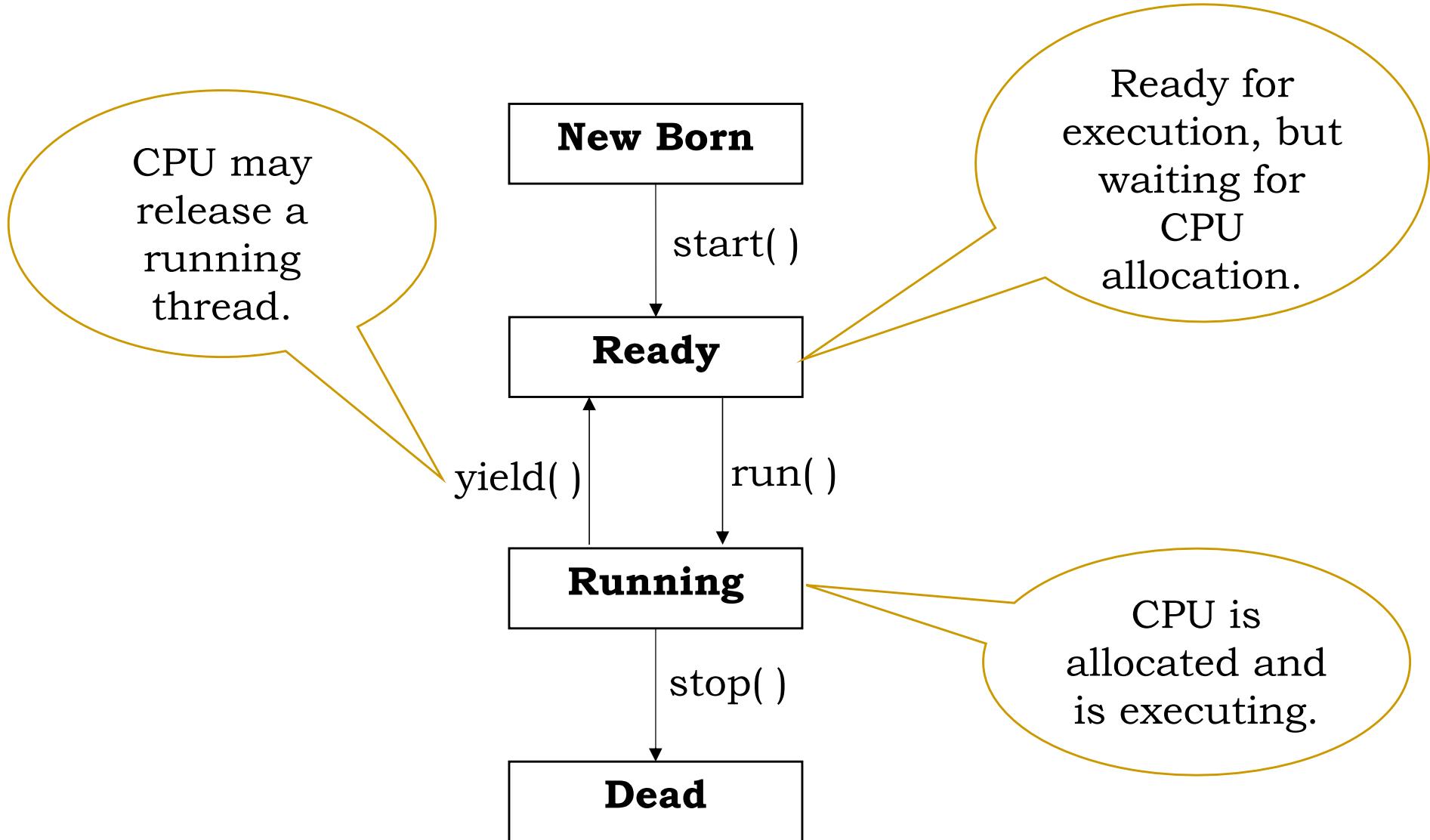
Single Thread

# Thread



Multithreading

# Life Cycle of a Thread



# Life Cycle of a Thread

## 1. NEW BORN

- ❑ A thread is started in this stage
- ❑ To start a thread we need to create an object
- ❑ Also allocate a working memory space for the thread
- ❑ This memory location is allocated within the memory for the process within which the thread works

# Life Cycle of a Thread

## 2. READY or RUNNABLE

- ❑ The thread is already created
- ❑ After memory allocation in “New Born” stage, the thread needs to be brought to the Ready state
- ❑ `start( )` is the only method that can be used for starting a new thread
- ❑ `start( )` brings a new born thread to a ready state
- ❑ Here the thread is ready to be executed, but have not been allocated any processor for execution (CPU unavailability).

# Life Cycle of a Thread

## 3. RUNNING

- ❑ run( ) method is used to allocate CPU to the thread
- ❑ The thread is in execution if run( ) method is executed properly
- ❑ run( ) method describes the task which the thread is going to perform
- ❑ Instructions written within the run( ) method describes what the thread is going to perform

# Life Cycle of a Thread

## 4. DEAD

- ❑ On completion of execution of instructions present in run( ) method
- ❑ Thread is moved to Dead state
- ❑ Otherwise we can use stop( ) method to stop execution of a thread
- ❑ stop( ) method can be used to move a thread from running state to dead state.
- ❑ A thread in dead state cannot be brought back to running or ready state.

# Thread program in Java

- Threads can be created in Java using;
  - Create a class which extends **Thread** parent class
  - Create a class which implements the **Runnable** interface
  
- Thread parent class
  - Primary class that helps to implement Multithreading in Java
  - `java.lang.Thread` => Thread class is present in `java.lang` package
  - Import `java.lang` to use Thread class
  - `start()` and `run()` methods are available within Thread class.

# Thread program in Java

## ■ Runnable interface

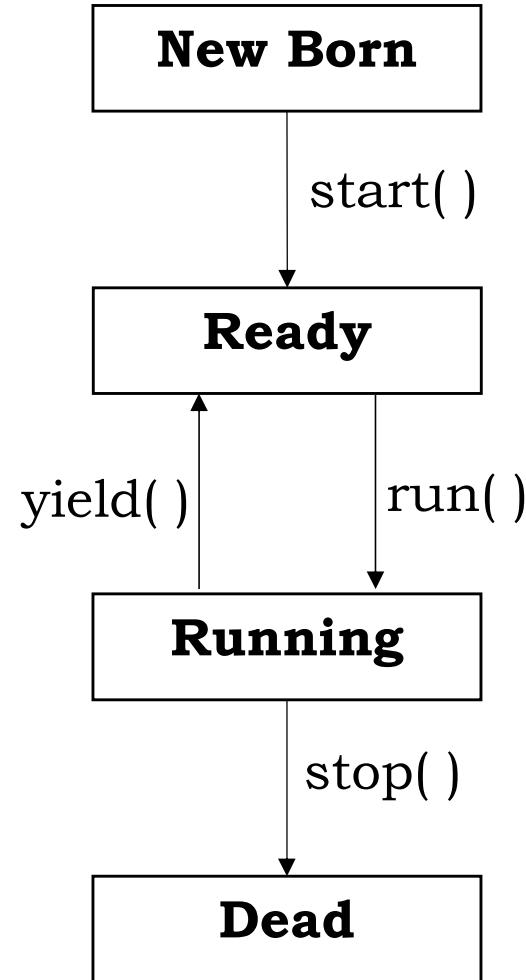
- Interface present within `java.lang` package
- Only abstract method `run()` is present
- No other methods present in `Thread` class is available here
- To start a thread using `Runnable` interface,
- We need to create object of `Thread` class
- Pass the object of the class along with the `Thread` object
- Use the `Thread` object to call `start` method.

# Thread programs

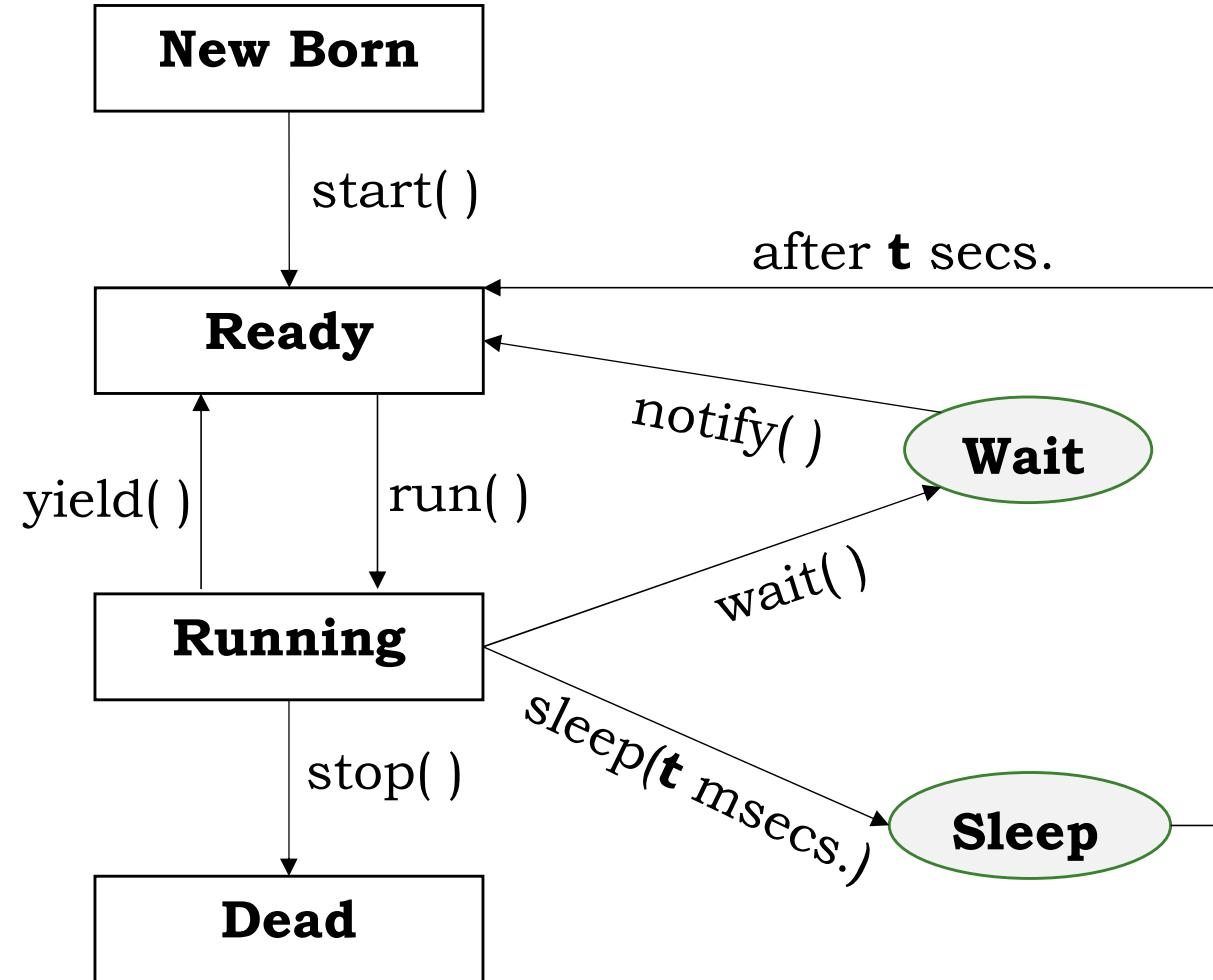
## ■ REFER PROGRAMS

- `thread_eg1.java`
- `thread_eg2.java`

# Life Cycle of a Thread



# Life Cycle of a Thread



# Life Cycle of a Thread

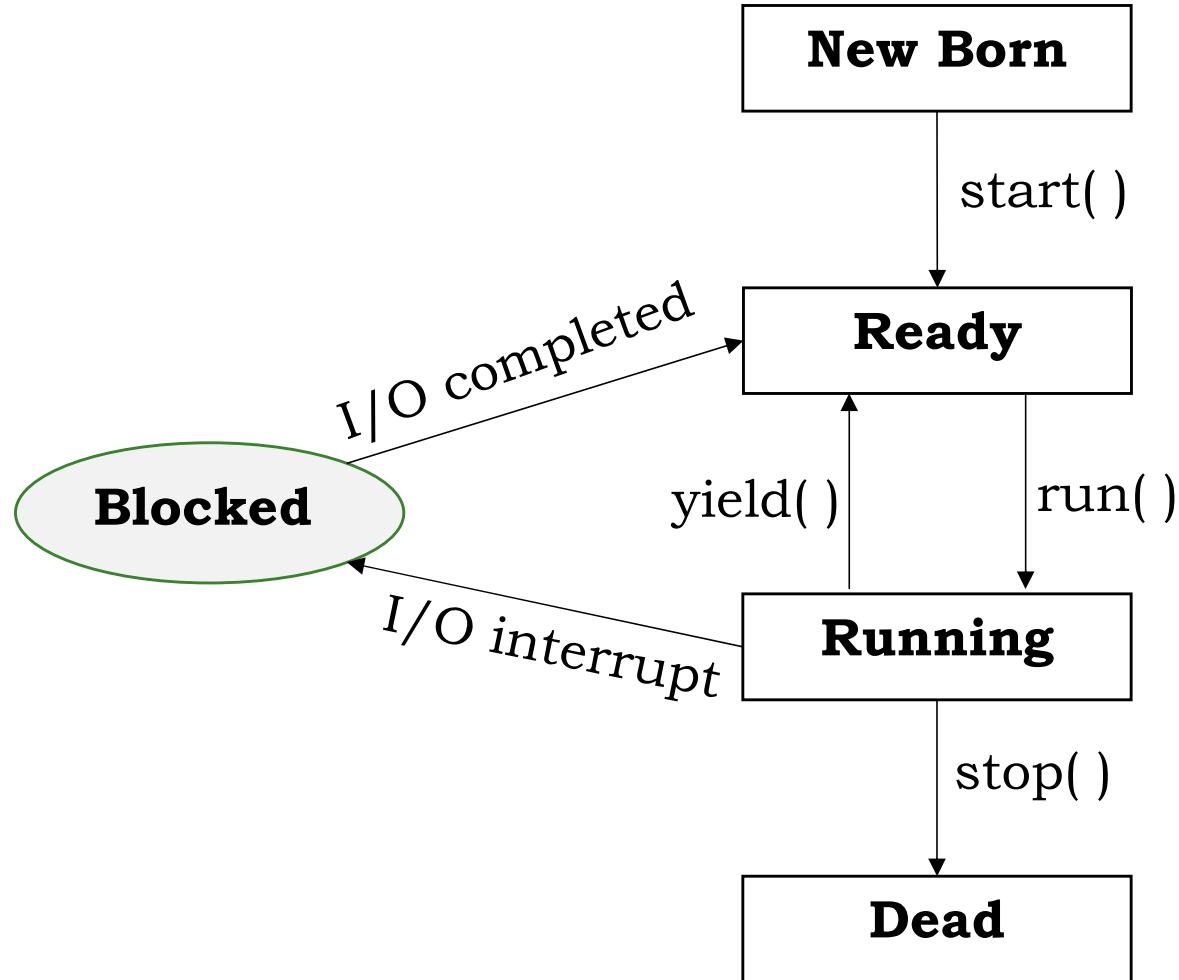
## ■ **sleep( )** [*paused execution*]

- Java pauses the thread for a specified seconds (user defined)
- Thread does not lose its ownership
- Thread resumes execution after the specified time is over
- CPU allocation is managed accordingly, so that after specified time it can be brought back to running state
- Failures of automatic transition from ready to running state may be disrupted in a few extra-ordinary scenarios.

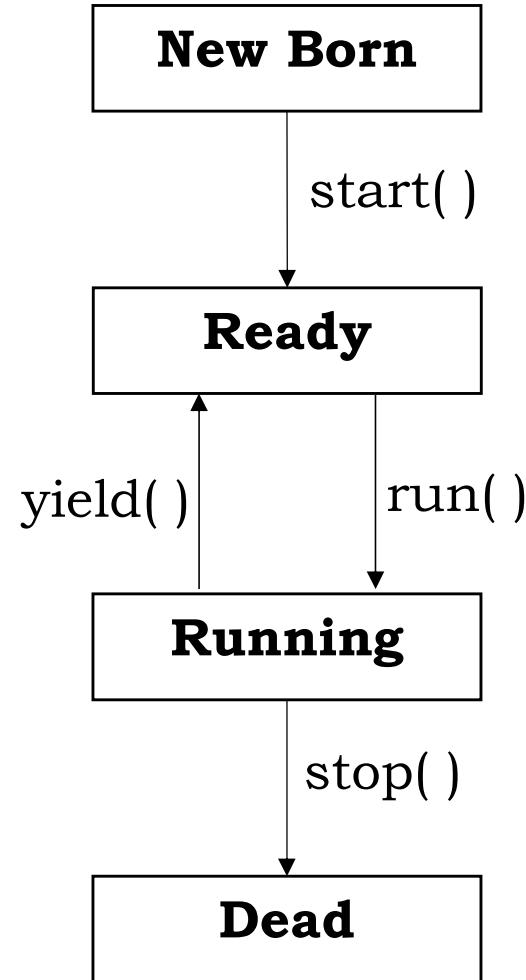
## ■ **wait( )** [*inter-thread communication*]

- Method is defined in `java.lang.Object` class
- Threads are not moved to ready queue directly on execution `wait()`
- This method makes a thread wait until the `notify()` method is called
- `notify()` method brings a thread from waiting state to ready state and wait for CPU allocation
- On CPU allocation the thread restarts execution.

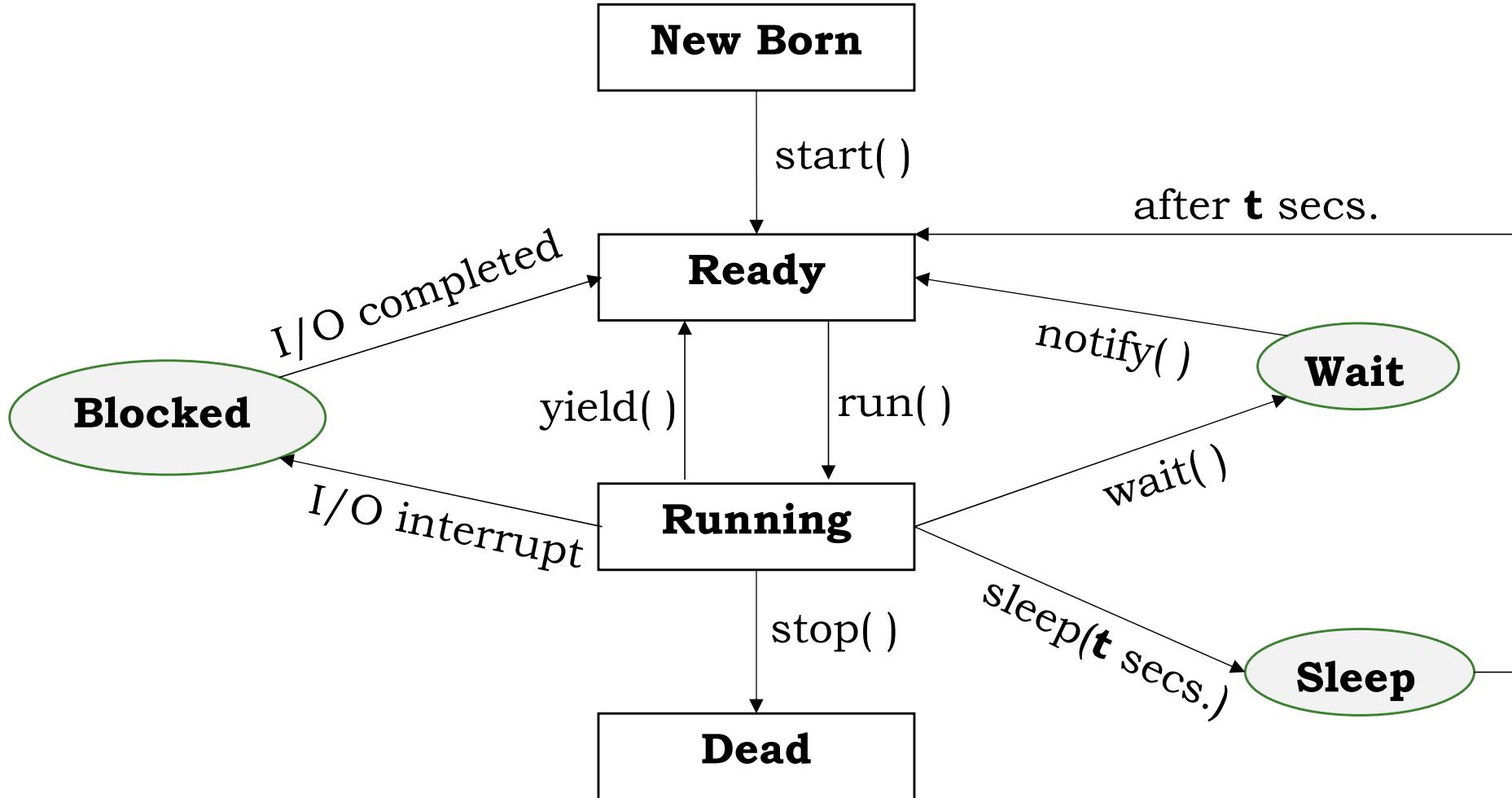
# Life Cycle of a Thread



# Life Cycle of a Thread



# Life Cycle of a Thread



# Thread programs

## ■ REFER PROGRAMS

- `thread_eg3.java`
- `thread_eg4.java`

# Thread Priority

- Priority to threads can be set
- Execution of threads are performed as per their priority
- Syntax:
  - `<ThreadName>.setPriority ( int Number )`
- Priority ranges between 10 to 1 (highest to lowest)
- 5 is the default priority
- `<thread name>.MAX_PRIORITY`, `<thread name>.MIN_PRIORITY`,  
`<thread name>.NORM_PRIORITY`
- `<thread name>.setPriority( )` , `<thread name>.getPriority( )`

# Thread programs

## ■ REFER PROGRAMS

- `thread_eg5.java`
- `thread_eg6.java`

# Synchronization

- A thread may try to read a resource which is now been used by another thread
- Thus depending upon the instructions in both the threads, the results may vary
- This is due to the fact that both the threads are accessing the same resource
- Synchronization technique is used in Java to address the problem
- If a method is declared to be **synchronized** then, whenever a thread calls this method for first time a **Monitor** is created over the method
- Thus other methods are restricted from calling the same method.

# Synchronization

- Therefore multiple threads may not access a single data/method at the same time

- Syntax:

```
synchronized void display( )  
{  
.....  
.....  
}
```

- **A monitor is an object that is used as a mutually exclusive lock or mutex**
- Monitor allows one thread at a time to execute a synchronized method on the object.

# Exception Handling

- Any error during program execution may produce incorrect output
- Even can terminate the program
- In some scenarios could halt the system
- Java provides a way to handle errors
- Types of errors;
  - Compile time error
  - Run time error
- Compile time errors:
  - Syntax errors in program coding
  - Detected and displayed on screen
  - Class file is not created in this scenario.

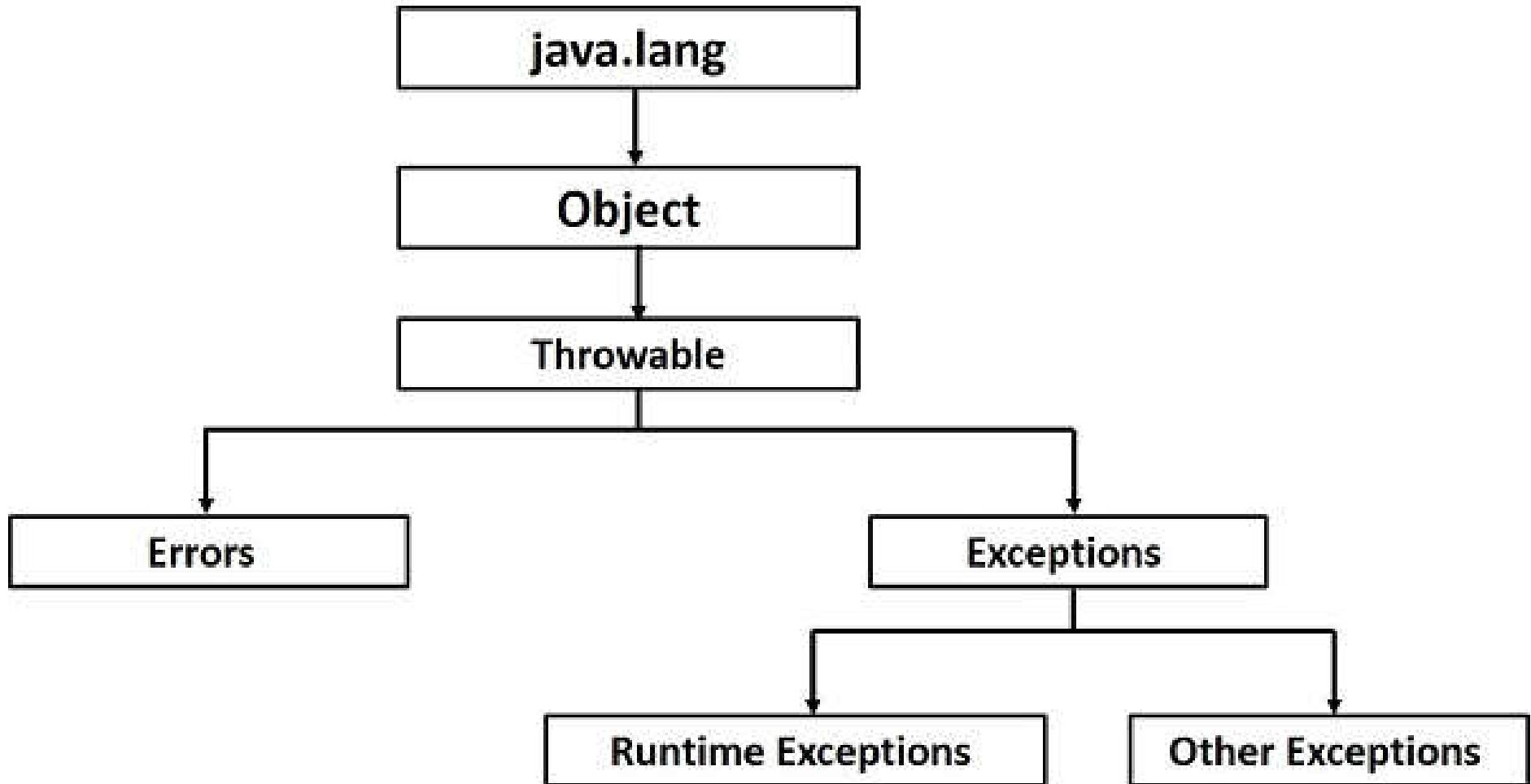
# Exception Handling

- Run time errors:
  - Programs may be compiled correctly and class file could be created
  - However, the program may fail to execute
  - Alternatively it may produce wrong results due to wrong logic in the programs
  - For example;
    - Dividing an integer by zero
    - Accessing an element which is out of an array limit, etc.
- Exception
  - Refers to an unwanted or unexpected event
  - That may occur during the execution of a program (runtime)
  - Disruption in program flow may occur
  - If such an exception is not handled then an error message is displayed
  - And the program is terminated
- It is desired to keep the program in execution and handle the unexpected event or exception.

# Exception Handling

<b>ERROR</b>	<b>EXCEPTION</b>
Lack of system resources and some scenarios related to the environment/resources in which the program is executing	The program code in execution generates an exception
Error cannot be recovered. Once occurred, the program execution is stopped	Exception can be handled and recovered. Does not affect program execution (if handled)
Errors are present in java.lang.Error package	Exceptions are present in java.lang.Exception package
Examples: Syntax errors, Logical errors, OutOfMemoryError, StackOverflowError, etc.	Examples: ArrayIndexOutOfBoundsException, ArithmeticException, IOException, etc.

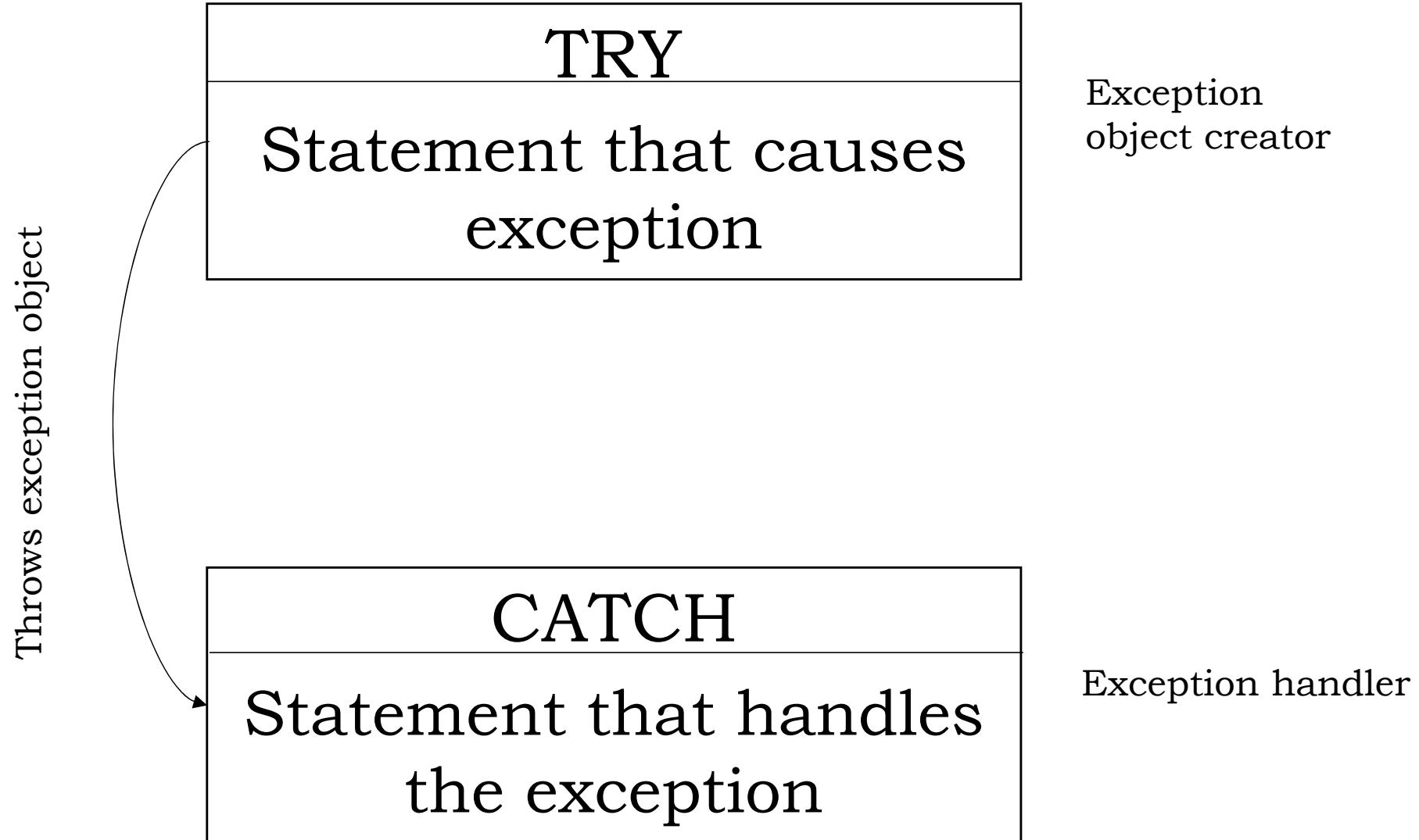
# Exception Class Hierarchy



# Exception Handling

- If we want to continue execution of the remaining code
- We must hold or catch the exception
- Subsequently display an appropriate message
- Continue execution of the rest of the program
- Exception handling tasks:
  - Find the problem (Hit the exception)
  - Inform error has occurred (THROW exception)
  - Receive error information (CATCH exception)
  - Take corrective action (Handle exception)

# Exception Handling



# Exception Handling

- Exception handling has two parts; TRY and CATCH
- Code which is likely to cause an error condition is kept in TRY
- Catch block handles the exception thrown by the try block
- Catch block also contains instructions for handling the exceptions
- If an exception occurs within the try block then
- Control jumps from try block to catch block
- Searches for the correct catch block that handles the occurred exception
- **Control never goes back from Catch to Try block.**

# Exception Handling

- Code written within the catch block is executed for handling the exception
- After execution, the control goes to the next line of the instruction after the Catch block
- Exception occurred only within a try block is handled in the corresponding catch block
- Exceptions created outside try block is never caught, and stops program execution.

# Exception Handling

```
try
{
    .....
    .....
}

catch(Exception_Type e)
{
    .....
    .....
    System.out.println( e.getMessage );
}

//Exception_Type is the type of exception to be handled

//Superclass for every exception is Exception
```

# Exception Handling

```
try
{
    .....
    .....
}

catch(Exception e)
{
    .....
    .....
    System.out.println( e.getMessage );
}

//All exception can be handled using Exception class
```

- Control from a Catch block never comes back to Try block.

# Exception Handling

- A try block can have multiple catch blocks
- The specific exception types are placed at the beginning
- Catch block with Exception class is kept at the last
- So that if specific exception is not handled then Exception class can handle
- There should not be any lines between TRY and CATCH
- **FINALLY BLOCK:**
  - Instructions written within Finally block is bound to execute
  - Instructions written within Finally block does not depend on whether exception occurred or not.

# Exception Handling

- Instructions which must be executed in a program might be kept in the Finally block

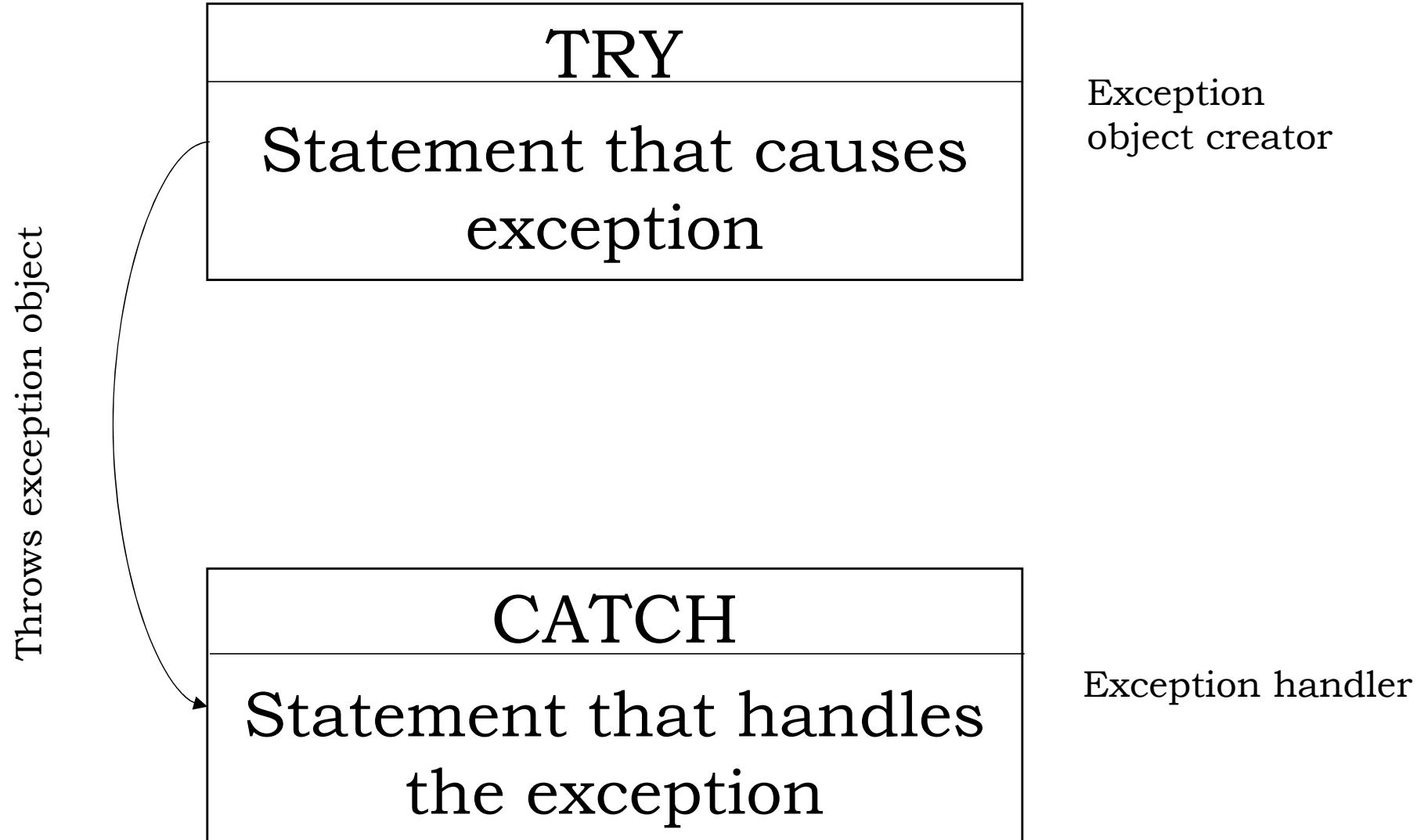
- Syntax:

```
try
{
    .....
}
catch(Exception e)
{
    .....
}
finally
{
    .....
}
```

# Exception Handling

- A message is generated by the JVM on encountering an exception
- The message contains the type of exception that occurred
- The line number where the exception has occurred, etc.
- **printStackTrace( )** method can be used to print this statement
- However, we can also provide our own customized message
- **Refer Program:** exception\_eg5.java

# Exception Handling



# Exception Handling

```
try
{
    .....
    .....
}

catch(Exception_Type e) {
    .....
    .....
    System.out.println( e.getMessage );
}

catch(Exception e) {
    .....
    .....
    System.out.println( e.getMessage );
}
```

# Throw Exception

```
public class exception_eg1 {  
    public static void main(String args[])  
    {  
        try  
        {  
            int x[] = new int[5];  
            x[20]=10;  
            System.out.println("Pramit");  
  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

# Throw Exception

- Usually exception is automatically identified and thrown by the JVM
- Unless it is manually identified and handled by programmer
- Throwing of exceptions can also be done manually
- Instructions to be displayed can be set according to requirements
- **Syntax:**

```
throw new ArrayIndexOutOfBoundsException ( "Here goes the Instruction" )
```

- **Refer program:** exception\_eg2.java

# Throw Exception

```
public class exception_eg2 {  
    public static void main(String args[ ]) {  
        try {  
            int x[ ] = new int[5];  
            x[4]=10;  
  
            throw new ArrayIndexOutOfBoundsException("Array is  
almost FULL !!");  
        }  
        catch(ArrayIndexOutOfBoundsException e)  
        {  
            e.printStackTrace( );  
        }  
    }  
}
```

# Throw User-defined Exception

- User-defined exceptions can also be created
- User-defined exceptions would have user-defined statements, which would be displayed to the user
- To manually **throw** a pre-defined exception we create an object of the required exception
- Syntax:
  - `throw new ArrayIndexOutOfBoundsException ( "Here goes the Instruction" )`
- However for user-defined exception we do not have a pre-defined class for which we can create the object
- Therefore, as the first step we would create class for the user-defined exception, that extends the Exception class
- Rest remains same. **Refer Program:** exception\_eg3.java

# Throw Exception

```
class      my_exception      extends  
Exception  
{  
    my_exception(String msg)  
    {  
        super(msg);  
    }  
}  
  
public class exception_eg2 {  
    public static void main(String  
args[ ]) {  
        try {  
  
int x[ ] = new int[5];  
x[4]=10;
```

```
        throw new my_exception("\n You  
are approaching the Array Limit !!  
"\n");  
    }  
    }  
    catch(my_exception e) {  
        System.out.println(e.getMessage( ));  
    }  
}
```

# Throws Exception

- Throw keyword can only be used if we want to handle an exception from the same method
- It is not possible that the exception occurs at one method and it is handled in another method: if we use Throw keyword
- Throws keyword can be used to address this problem
- Usual Scenario:
  - Class 1 with Method 1 and Class 2 with Method 2
  - Method 1 throws an exception
  - That exception is handled in Method 1
  - Method 2 throws exception which is handled in Method 2

# Throws Exception

## ■ **New Scenario:**

- Class 1 with Method 1 and Class 2 with Method 2
- Method 1 throws an exception
- That exception is handled in Method 2
- Method 2 throws an exception
- That exception is handled in Method 1

## ■ **Refer Program:** exception\_eg4.java

# Throw Exception

```
class test_throws
{
    void         disp()      throws
ArithmeticException, Exception
    {
        int x = 7/0;
    }
}

public class exception_eg4
{
    public static void main(String args[])
    {
        test_throws T = new test_throws();
    }
}
```

```
try
{
    T.disp( );
}

catch(ArithmeticException e)      {
    e.printStackTrace( );
}

catch(Exception e)  {
    e.printStackTrace();
}
}
```

# Elements of Object Models

- Four major elements of the Object Model
  - *Abstraction*
  - *Encapsulation*
  - Modularity
  - Hierarchy
- Three minor elements of the Object Model (useful but not essential)
  - Typing
  - Concurrency
  - **Persistence**

# Persistence

- Persistence is the property of an object through which its existence transcends time
- Once an object is created
  - It occupies certain memory space
  - It stays for a certain time
- Persistence is the relationship of an object with “time” and “space”
- Time: Object continues to exist after its creator ceases to exist
- Once the creator no longer exists then how long will the object stay or exist is an independent decision taken by OS
- Space: Object’s location moves from the address space in which it was created.

# Continuum of Object Existence

- An object in software takes up some amount of space and exists for a particular amount of time
- There is a continuum of object existence, ranging from **transitory objects** that arise within the evaluation of an expression to **objects in a database** that **outlive** the execution of a single program
- **Transitory objects:** objects that are created temporarily, for an interim operation. Example: return a variable with the value and store the returned value within the original variable of the program. So the variable which brings the value from the called function to the calling function is temporary/ interim object and we do not need it later
- **Objects in DB/Persistent Objects:** An object/ data stored in DB for representing an employee. Now the employee may have left. The operation/interface/functionality given to that user, might be removed as he/she has left. However, the object pointing to that employee may remain for more duration (outlive).

# Continuum of Object Existence

- Spectrum of object persistence: EXAMPLES
  - Transient results in expression evaluation
  - Local variables in function invocation
  - Global variables and heap items whose extent is different from their scope
  - Data that exists between executions of a program
  - Data that exists between various versions of a program
  - Data that outlives the program.

# Why Persistence is Important?

- If an object is very transitive, i.e. it occurs and then disappears then we would like to use a lightweight process for the task/object
- If not then system would have a lot of overhead in creating and destroying the objects
- If an object is very large and takes a lot of space then we must ensure that it is having more persistent behaviour than transitive behavior
- Large objects or objects in DB which is regularly used (for storage or relocation) should be treated separately so that we do not have system performance issues.

# Garbage Collection

- JVM maintains a heap to store all objects created by a running java application
- Garbage collection is the process of automatically freeing objects that are no longer referenced by the program
- This is also called memory recycling theory
- When an object is no longer referenced/ used by the program, the heap space it was occupying can be used by other new objects
- Very effective for memory management
- Garbage collector determines the object which is not referenced by the program for a certain duration of time.

# Garbage Collection

- Garbage collector is a part of the JVM which significantly helps in memory management related to Java programs
- Programmers do not need to explicitly identify objects which are currently not in use
- Garbage collector ensures program integrity
- Programmers do not have the control to delete an object from memory
- This ensures incorrectly freeing of memory and thus preventing accidental crash of JVM
- Garbage collection is executed using a Daemon thread, which always runs in the background.

# Finalize

- Finalize is a method present in `java.lang.Object` class
- Every class by default inherits the `finalize()` method
- The daemon thread for garbage collection always runs in behind
- Garbage collection process automatically identifies the un-used objects and free them from memory
- Programmers are not notified when an un-used object is being deleted from memory
- However, we can programmatically provide a notification system which would be invoked on deletion of an object.

# Finalize

- finalize( ) method can be used for this purpose
- A garbage collector before deleting an object searches for finalize( ) method if present
- If present, the method is executed and customized instruction can be provided to the programmer
- If any exception is thrown by the finalize method then the entire finalization process is blocked.
- **REFER PROGRAM:** finalize\_eg.java

# Finalize

```
class test_finalize {  
    protected void finalize( ) {  
        System.out.println("Object Deleted !!");  
    }  
}  
  
public class finalize_eg {  
    public static void main(String args[ ]) {  
        for(int i=0; i<5; i++)  
        {  
            test_finalize x = new test_finalize();  
        }  
  
        System.gc( );  
    }  
}
```

# Sequence Diagram

## What is a Sequence Diagram?

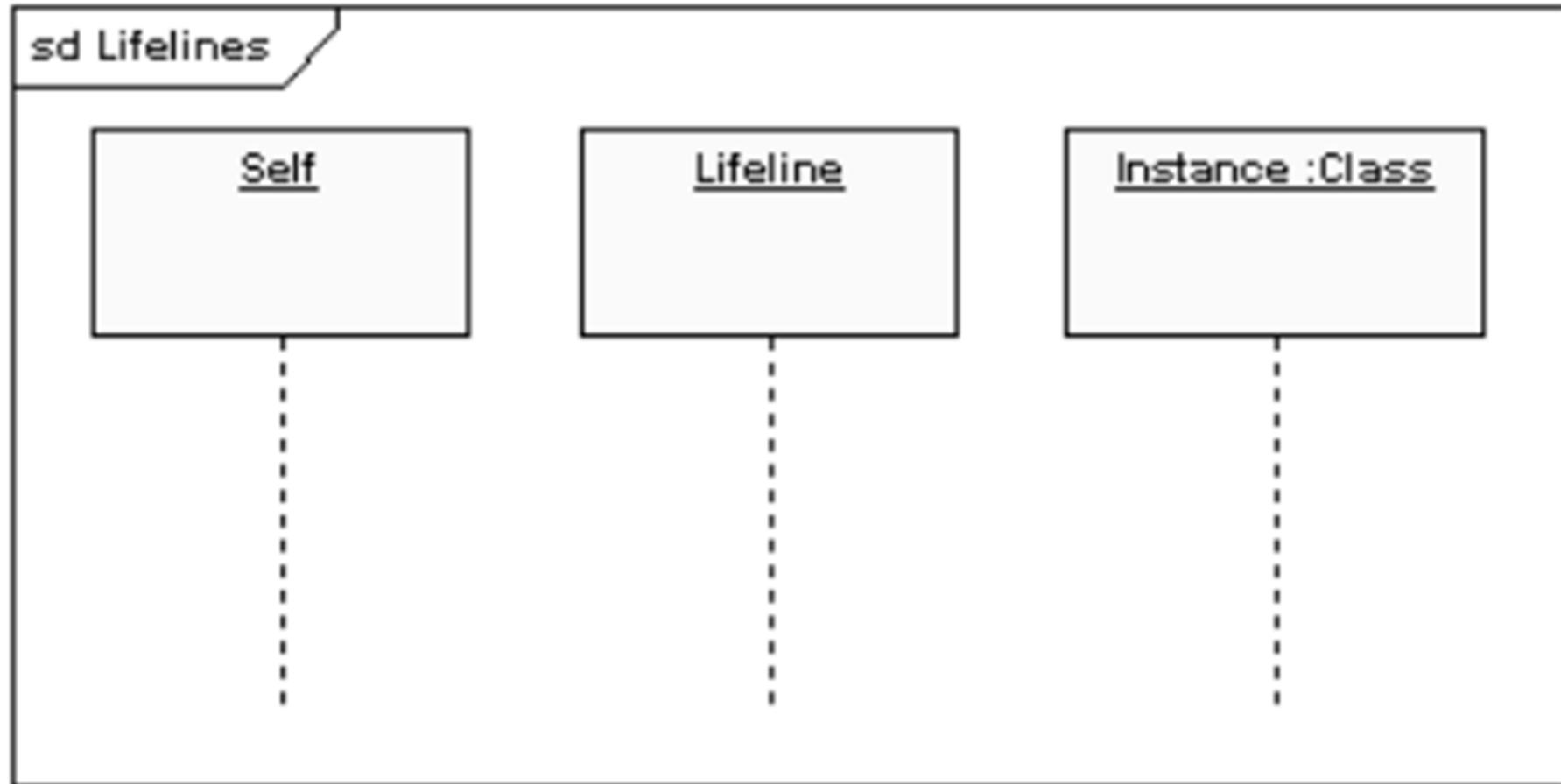
- A sequence diagram shows object interactions arranged in time sequence
- It depicts the object and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario
- Typically, a sequence diagram captures the behavior of a single activity or a use case

# Sequence Diagram

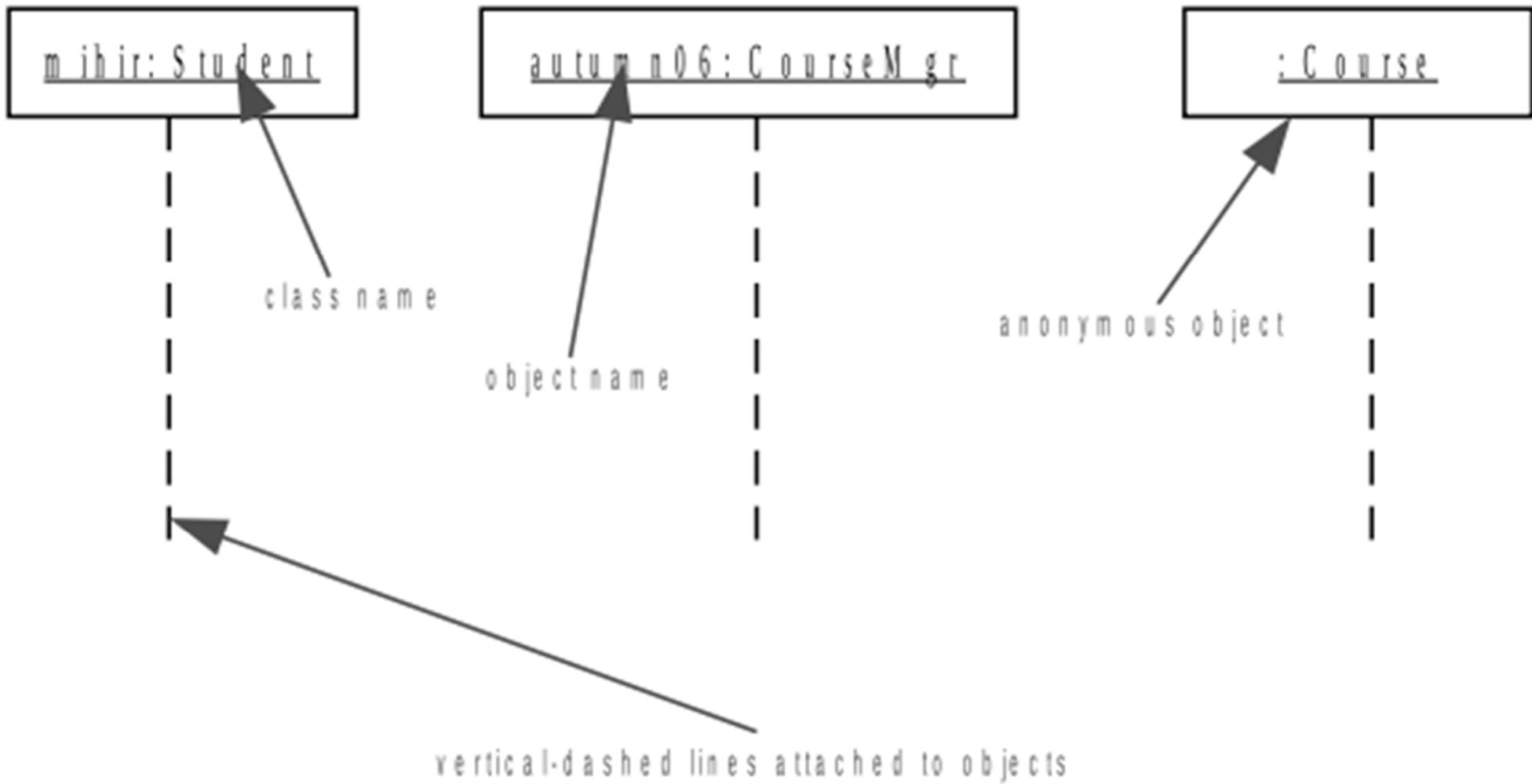
## Basic of a Sequence Diagram

- A sequence diagram is a two dimensional chart
- The chart is read from top to bottom
- The objects participating in the interaction are shown at the top of the chart as boxes attached to a vertical-dashed line
- Inside the box the name of the object is written with a colon separating it from the name of the class and both the name of the class and object are underlined
- Sometimes an anonymous object (only class name and underlined) is also used

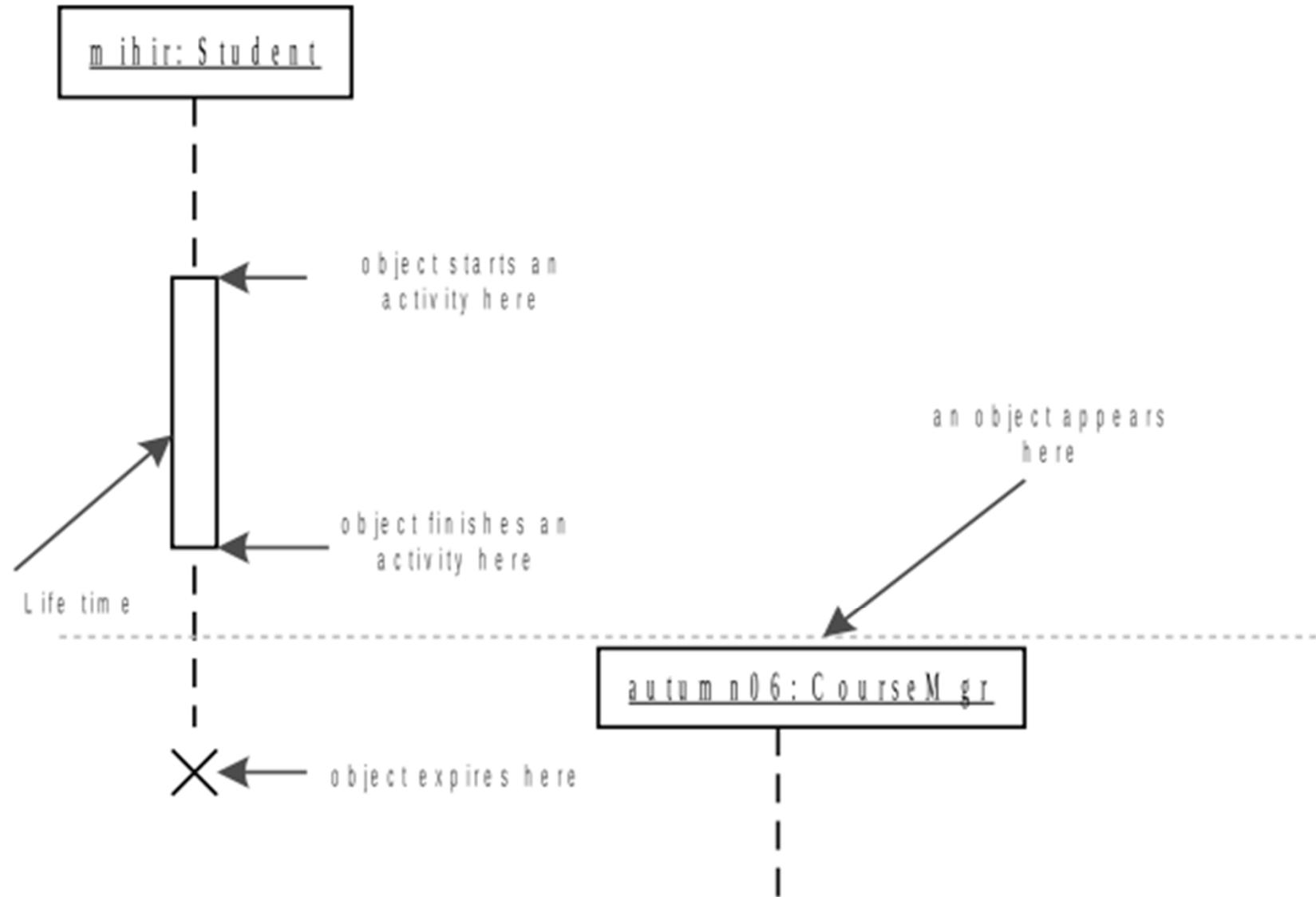
# Lifeline in a Sequence Diagram



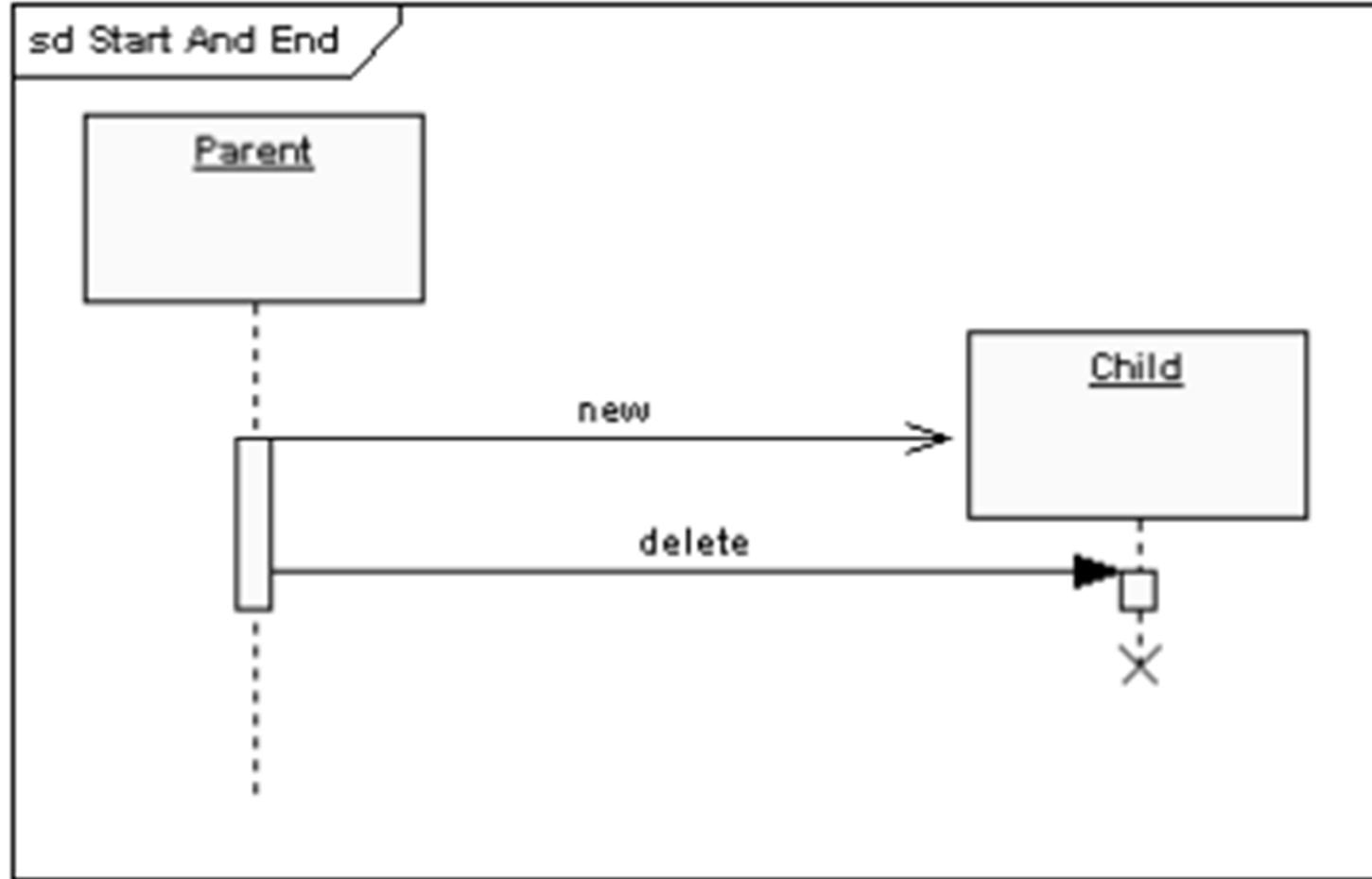
# Objects and Lifeline



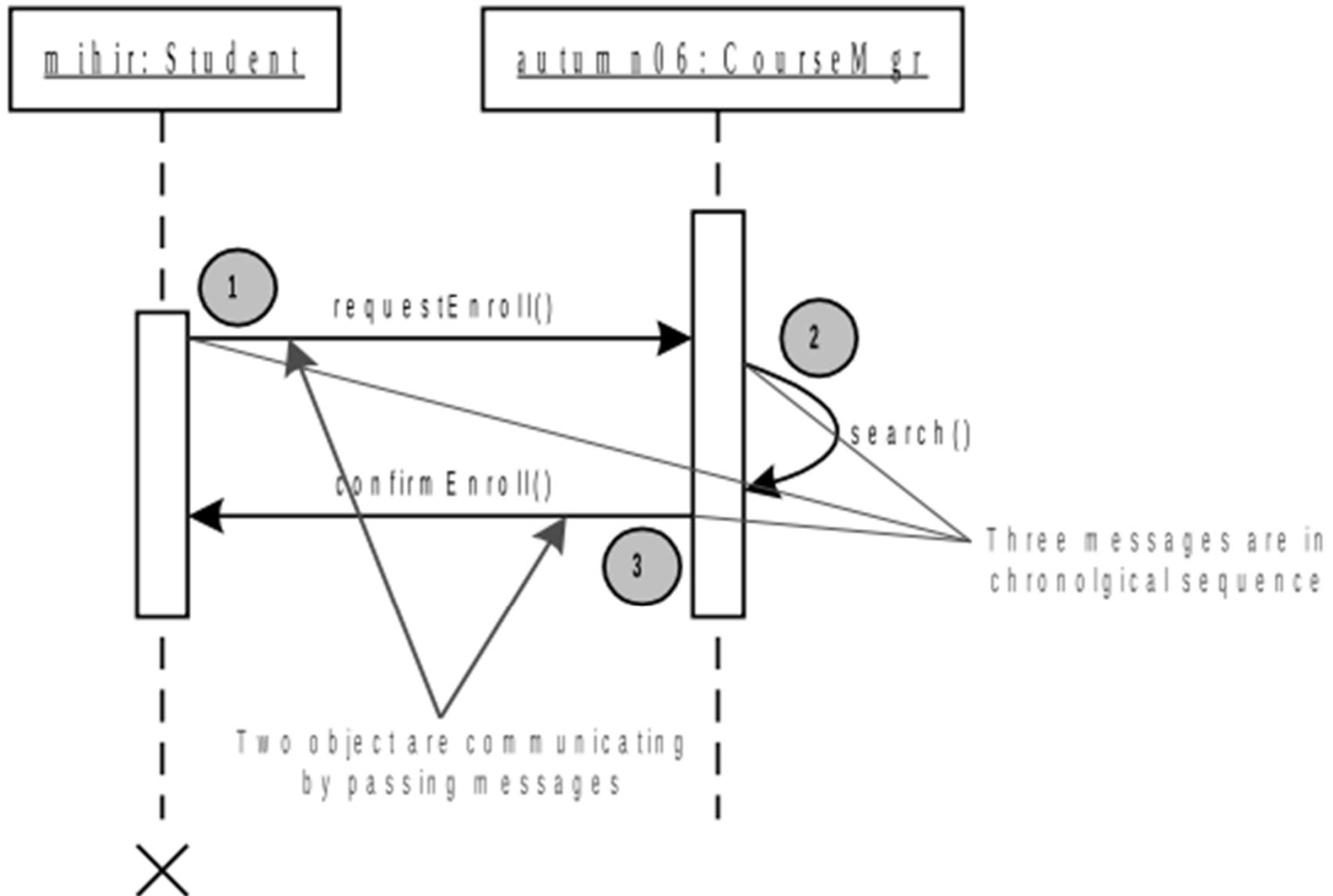
# Objects and Life Time



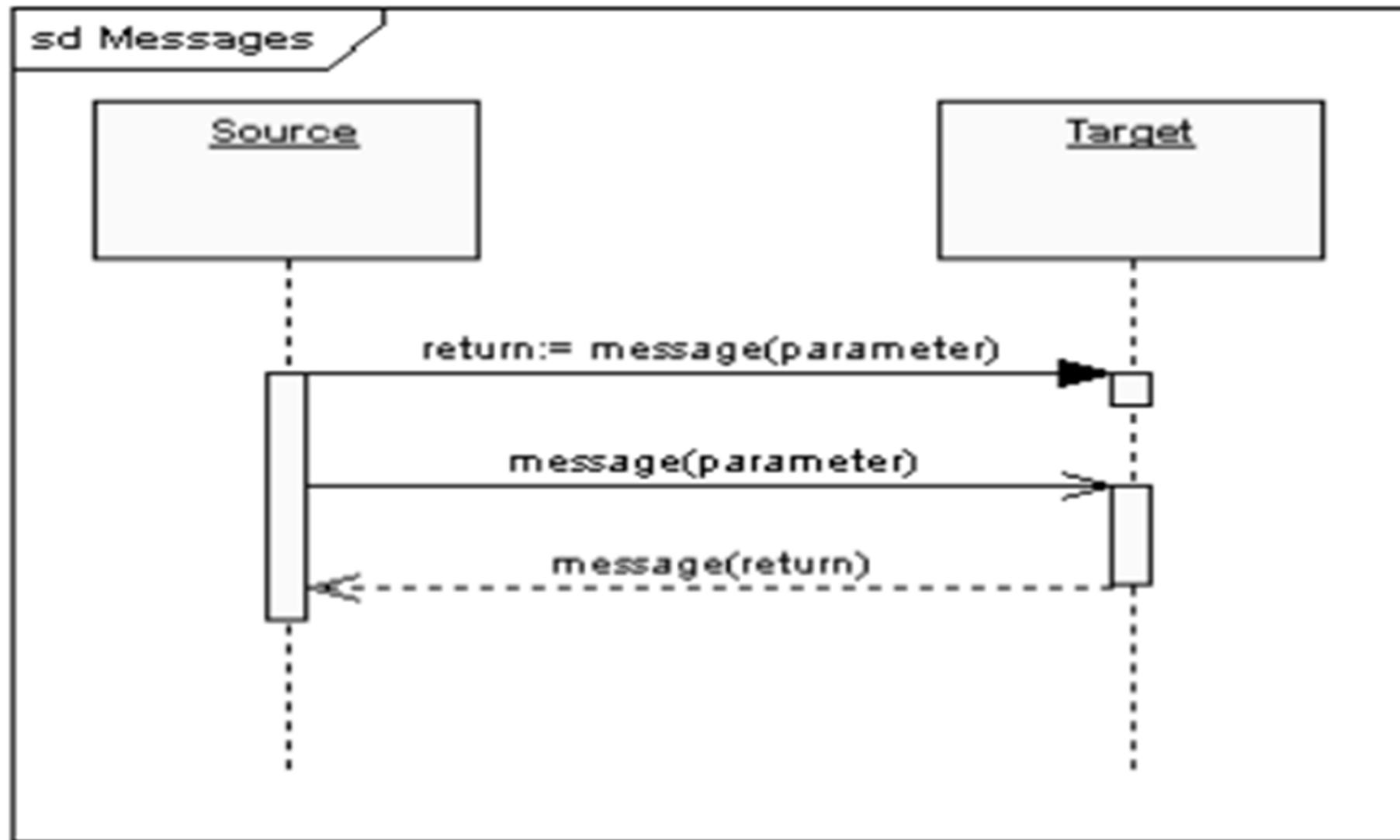
# Start and End of Life Line



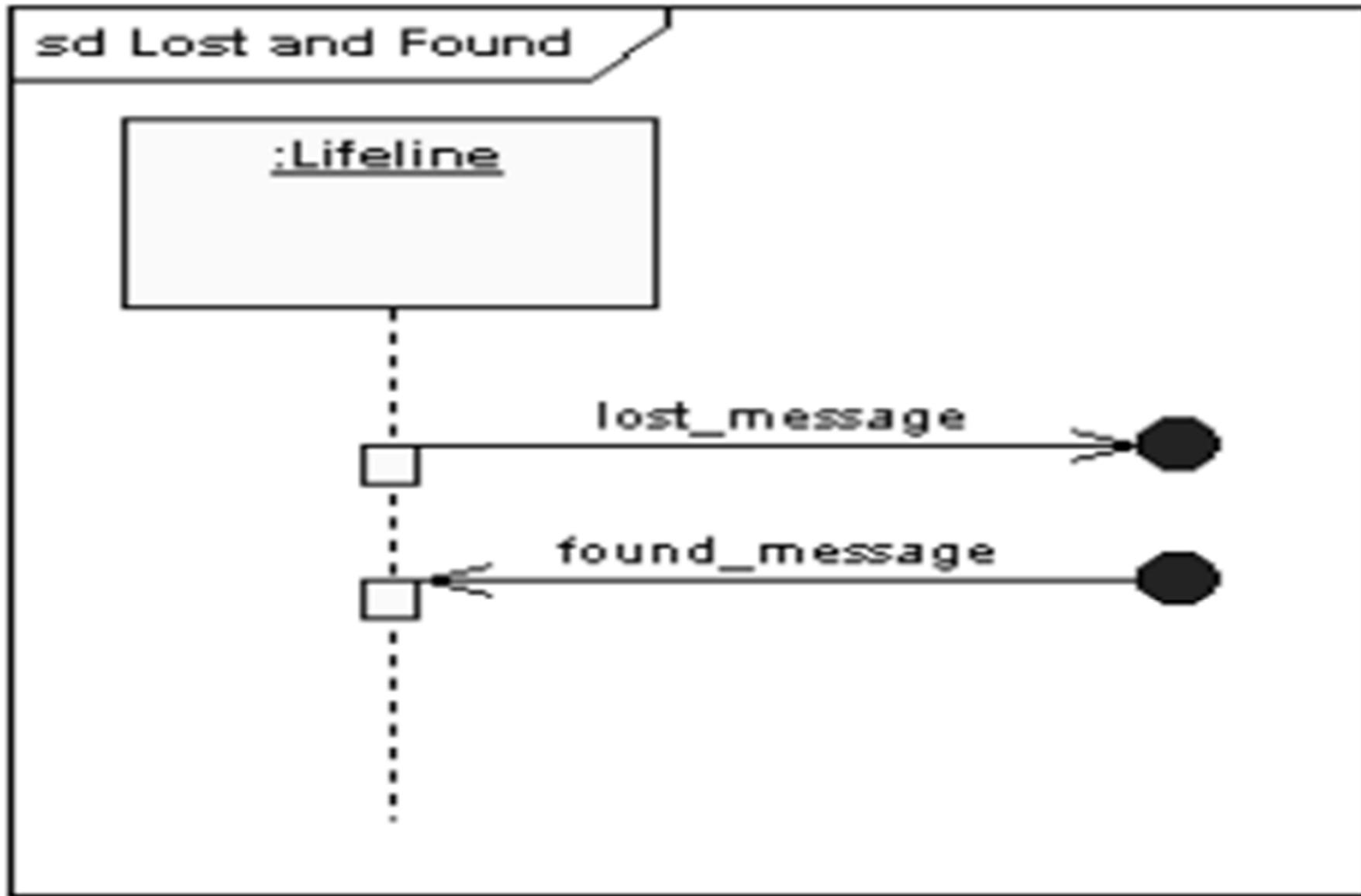
# Messages in Sequence Diagram



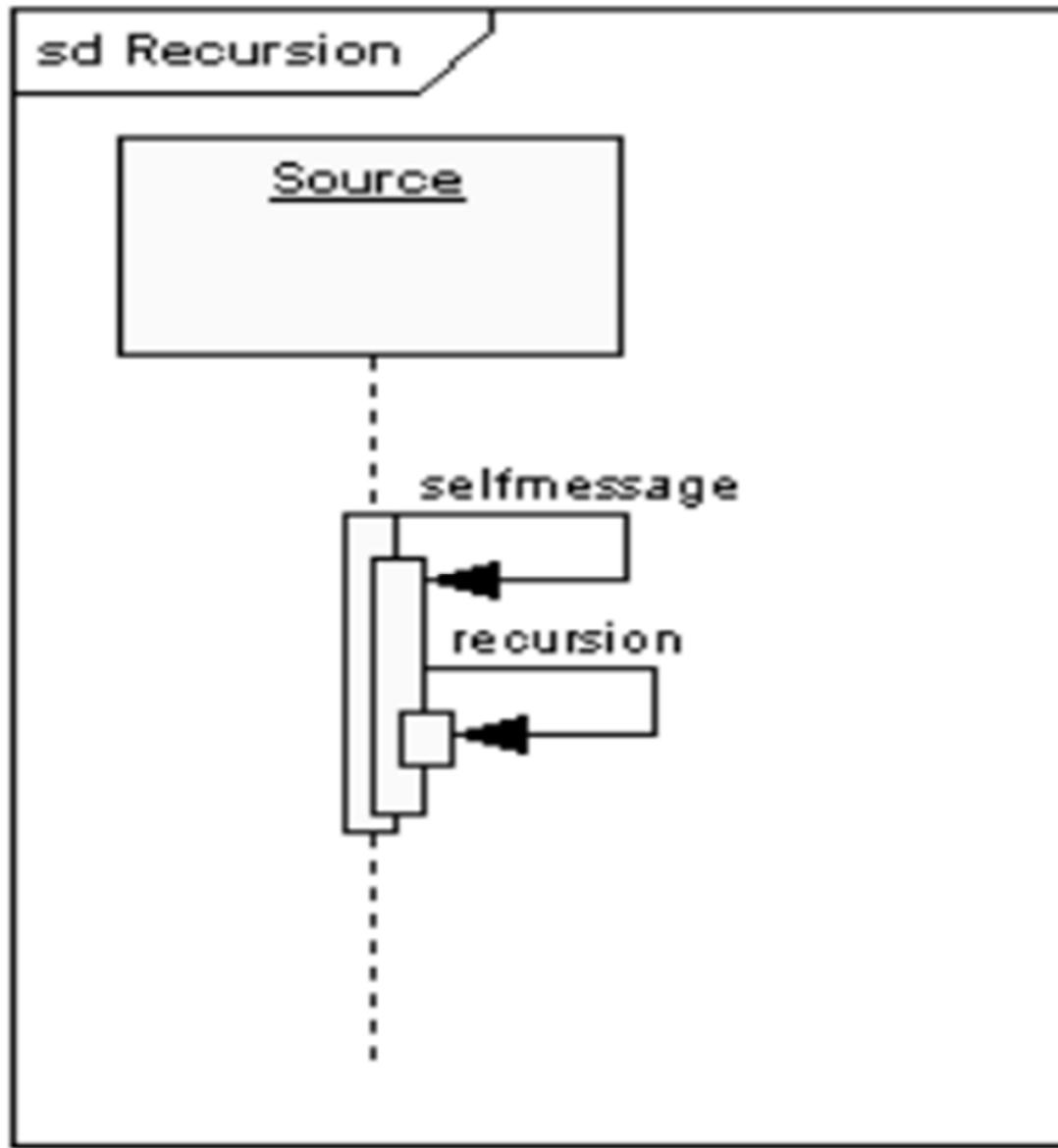
# Synchronous and Asynchronous Messages



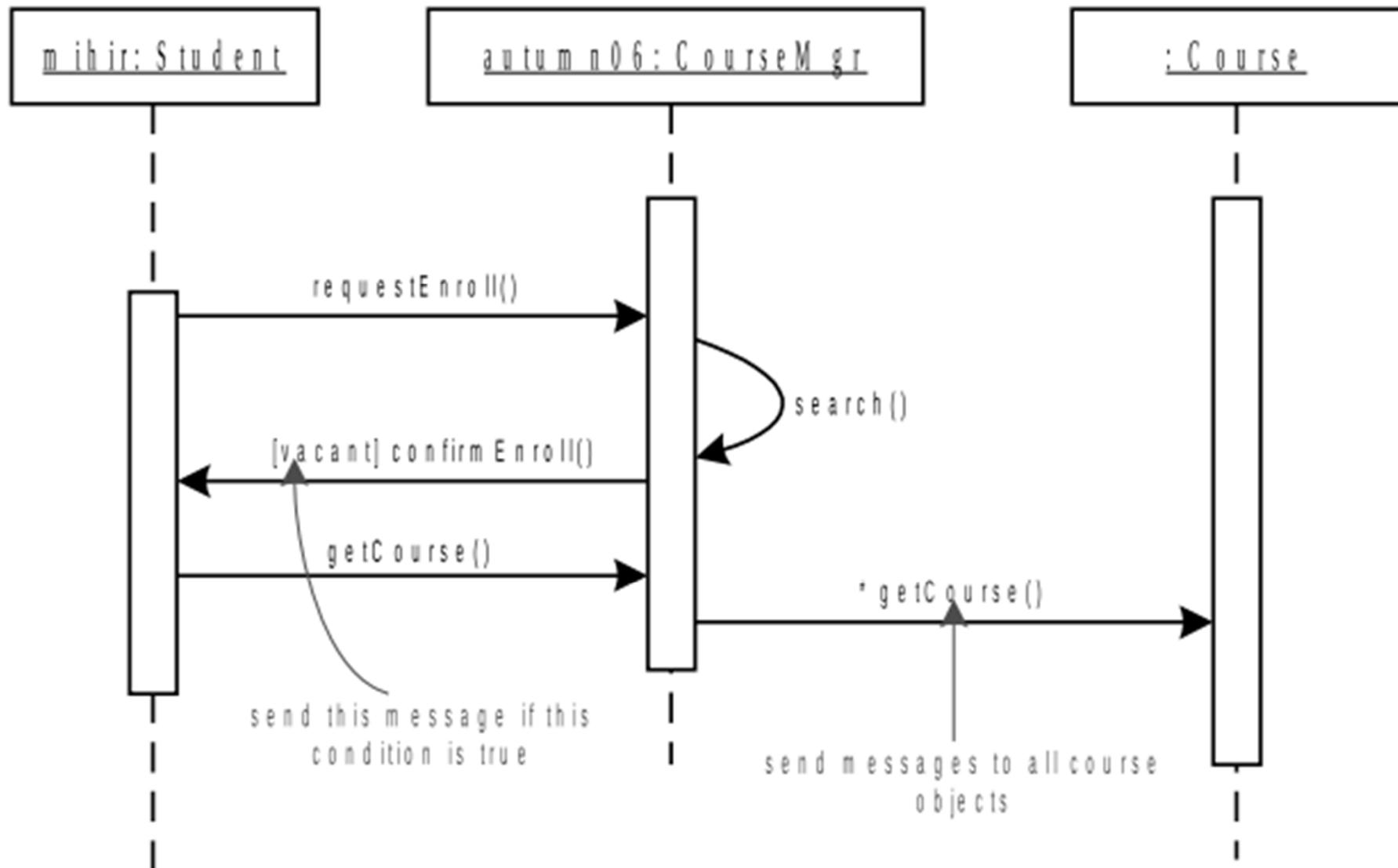
# Lost and Found Message



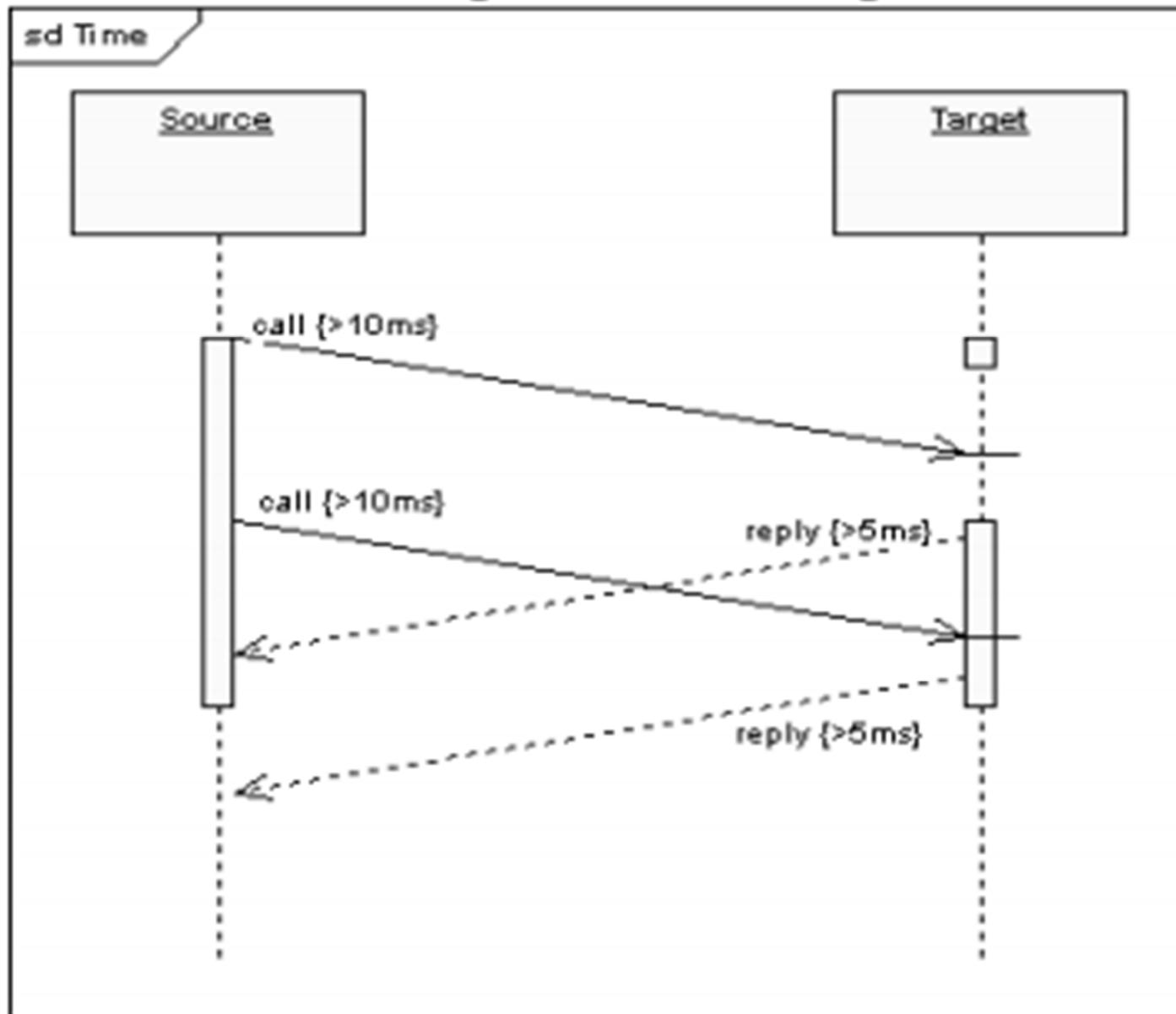
# Self or Recursive Message



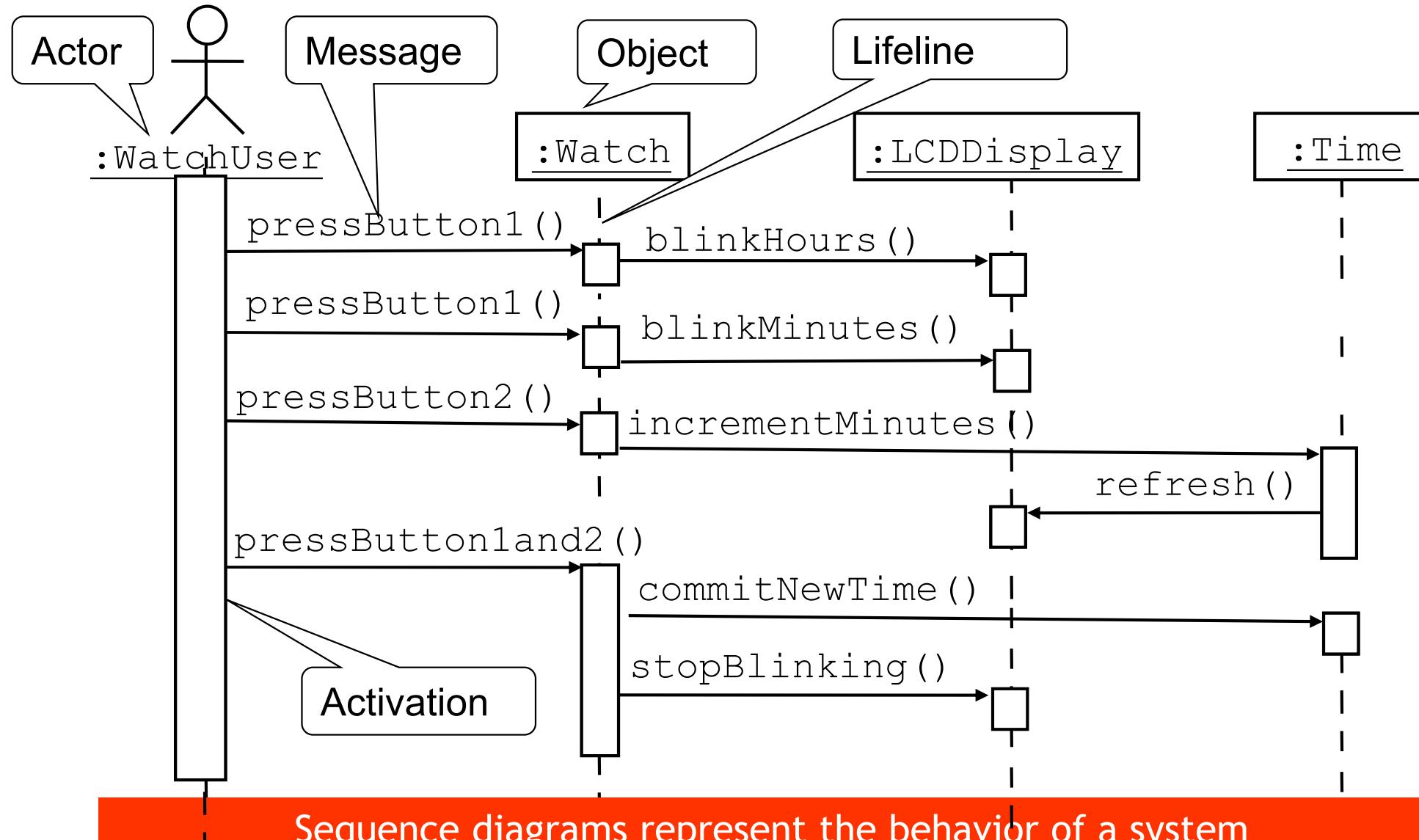
# Controlled Message



# Duration and Timing Constraints

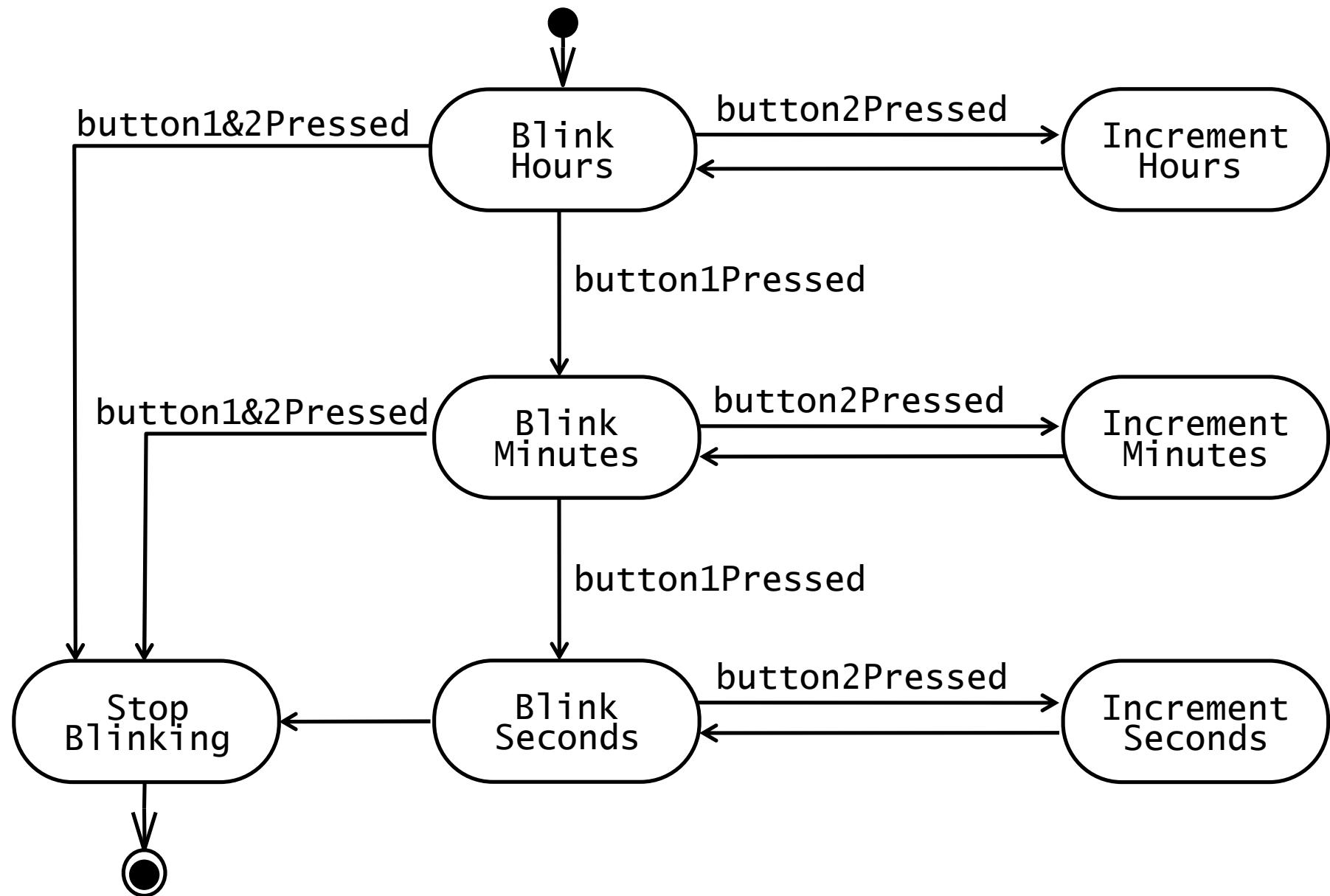


# UML: Sequence diagram

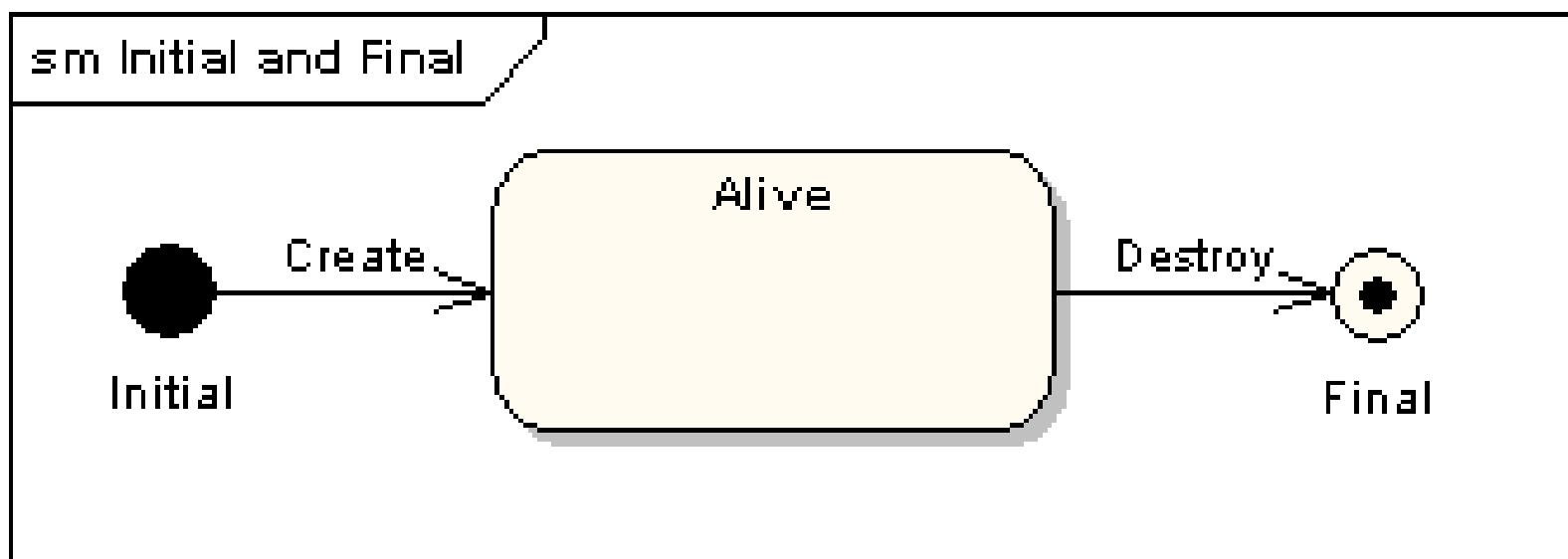
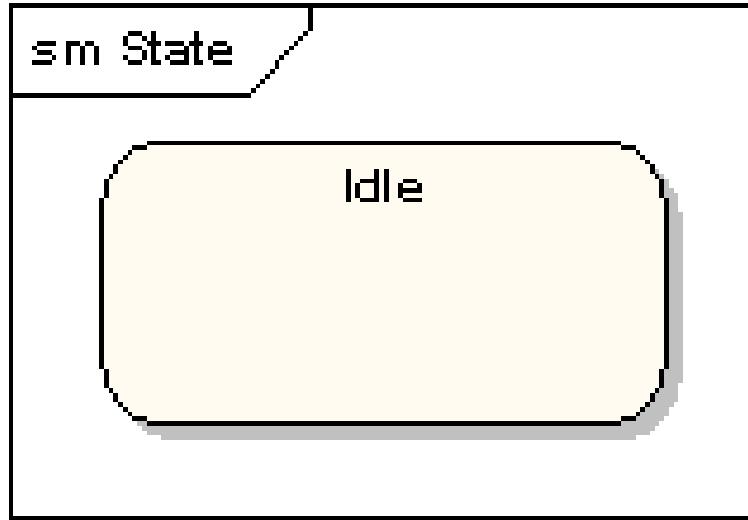


Sequence diagrams represent the behavior of a system as messages (“interactions”) between *different objects*

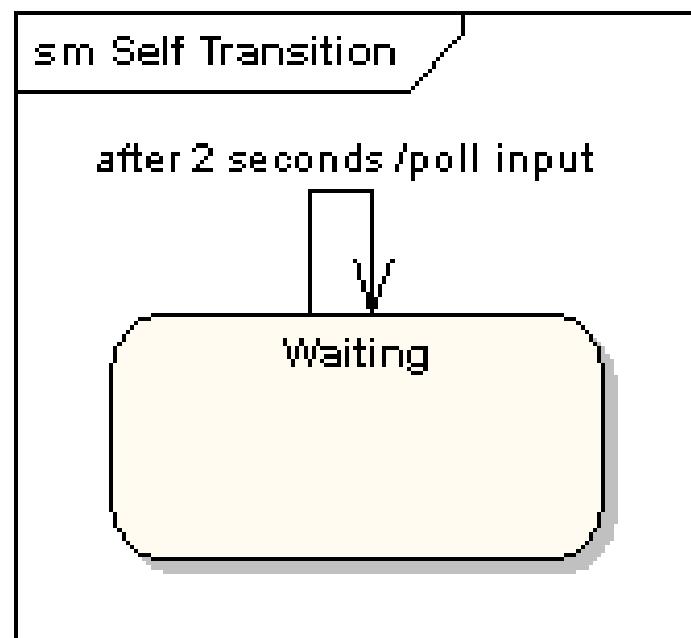
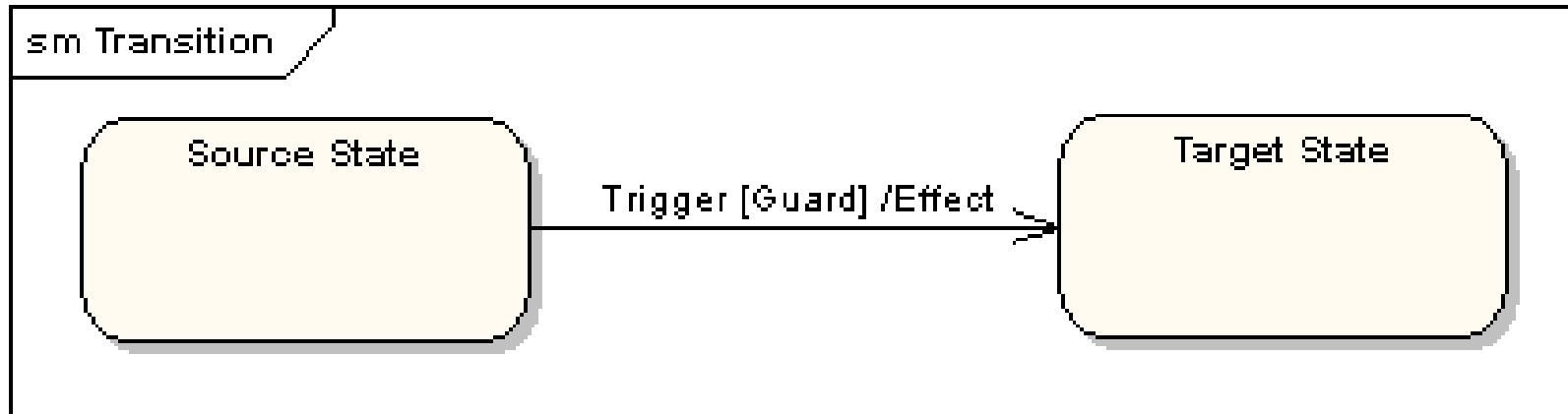
# State Machine Diagram \_ Sample Example



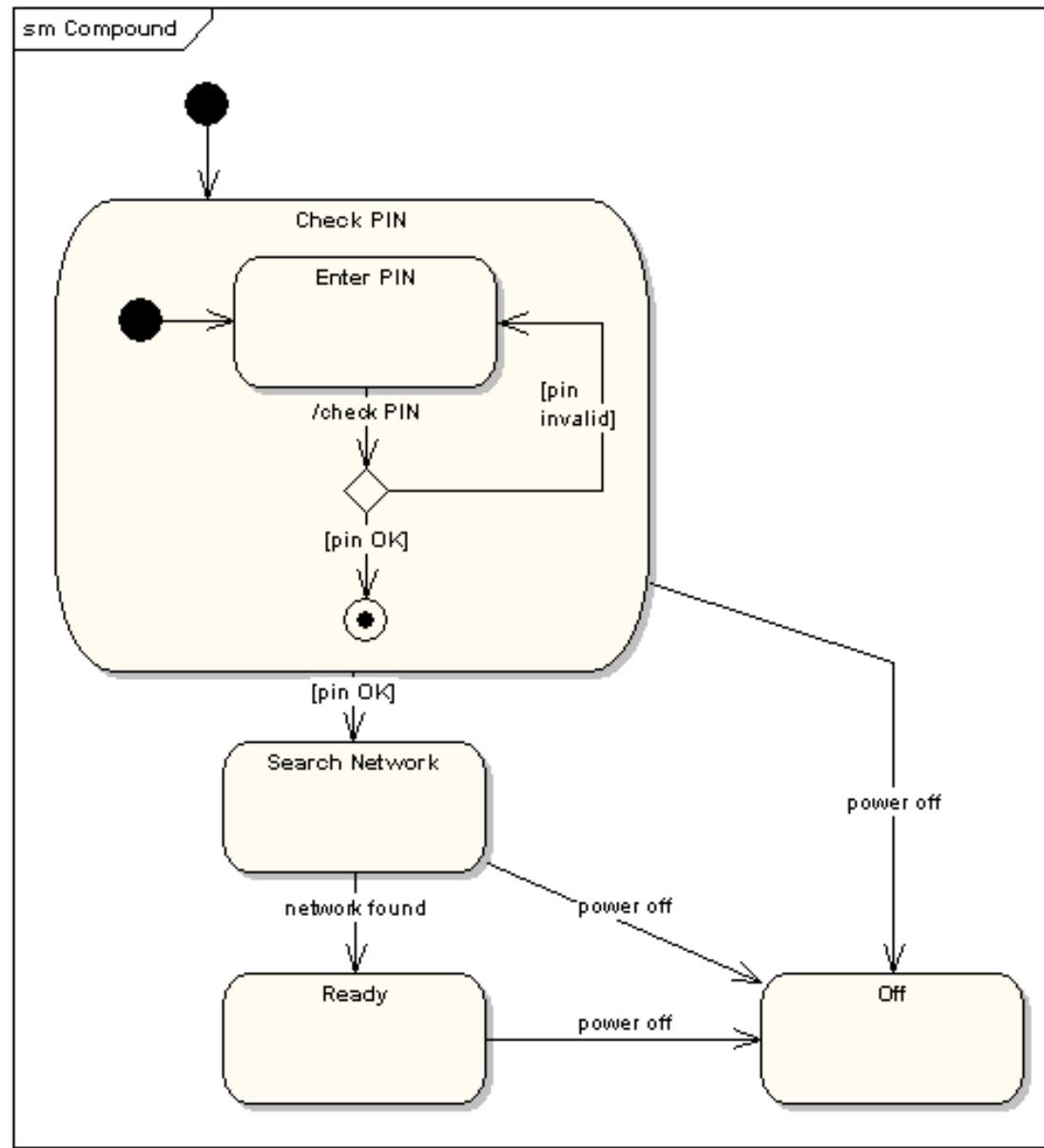
# State Machine Diagram \_ States



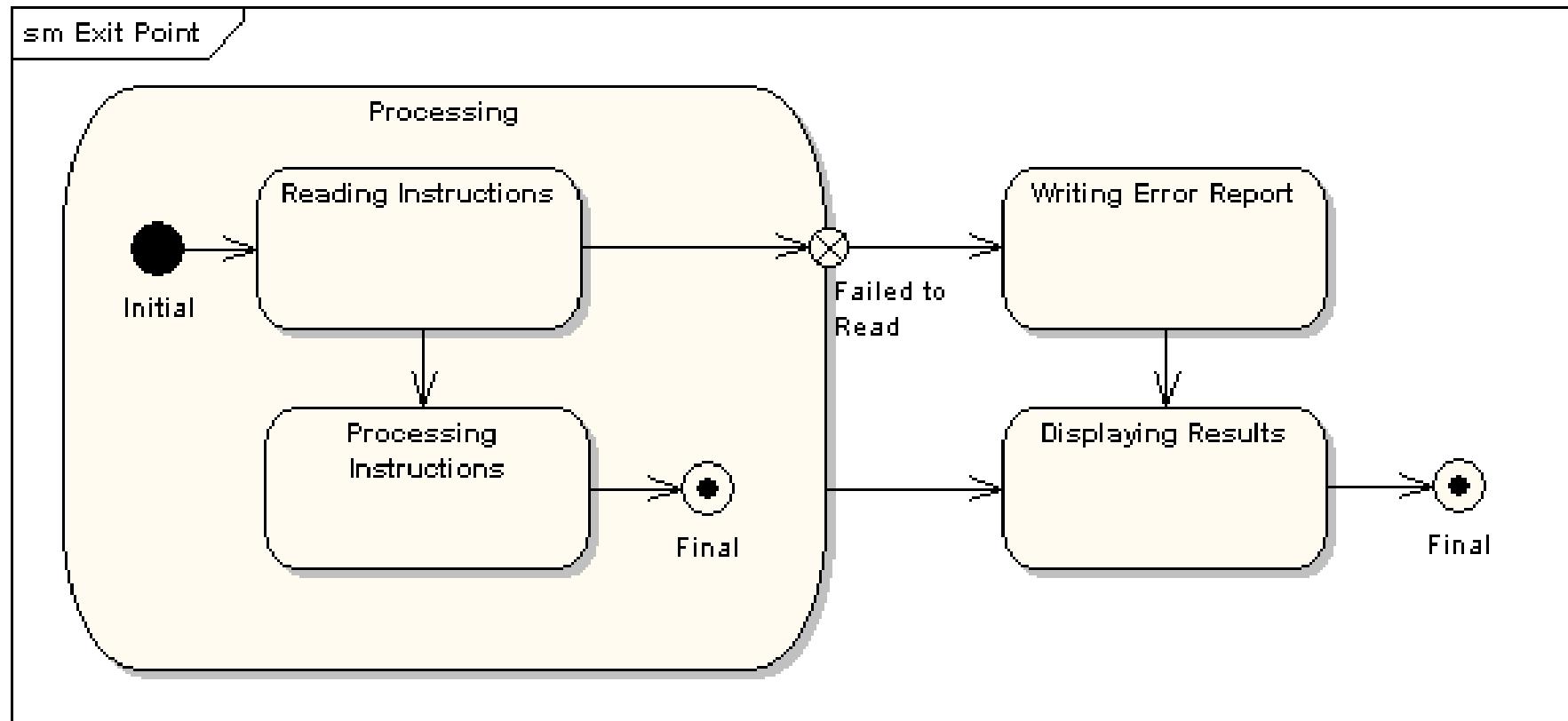
# Transition \_ Self Transition



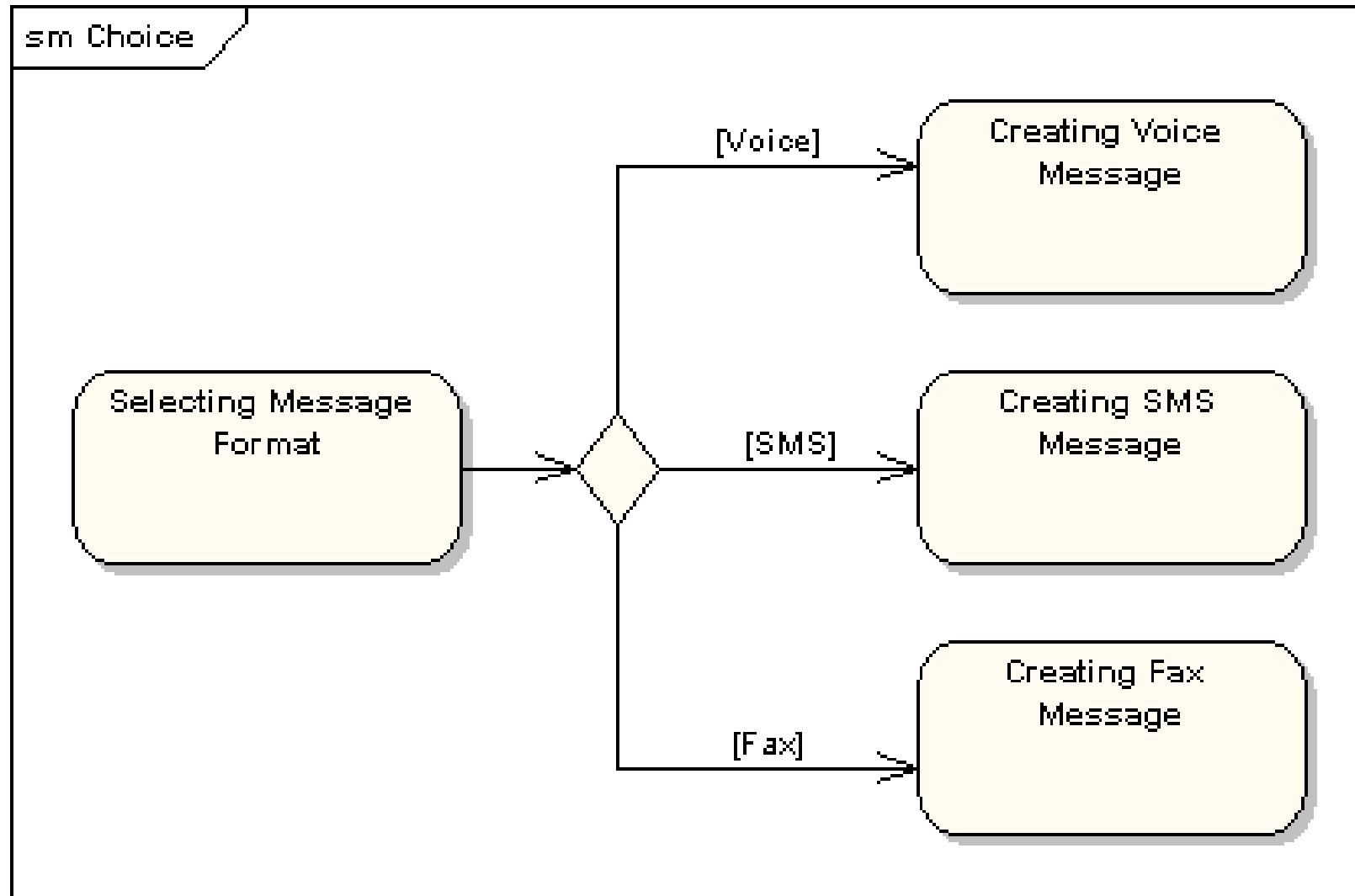
# Compound State



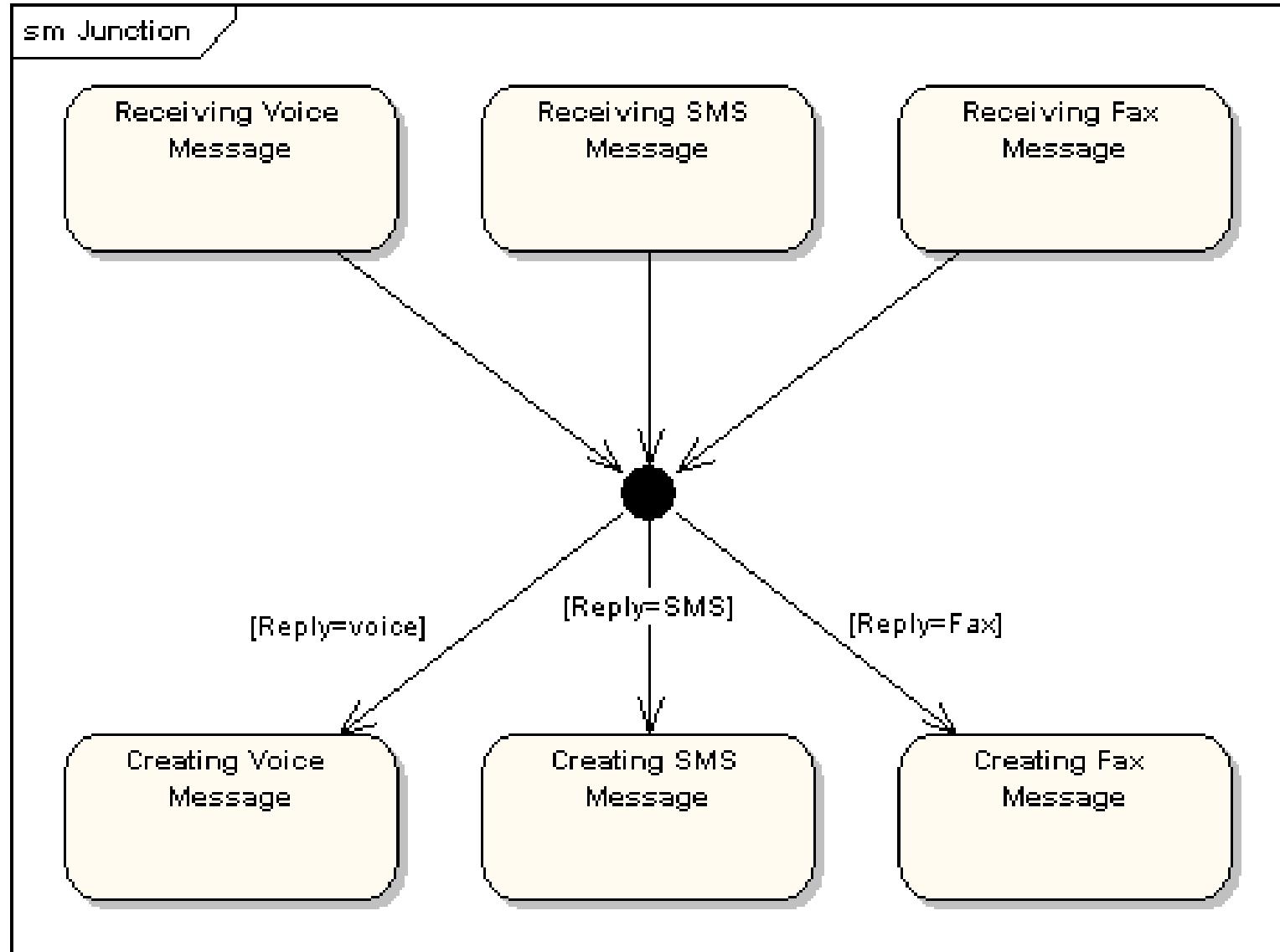
# Entry \_ Exit States



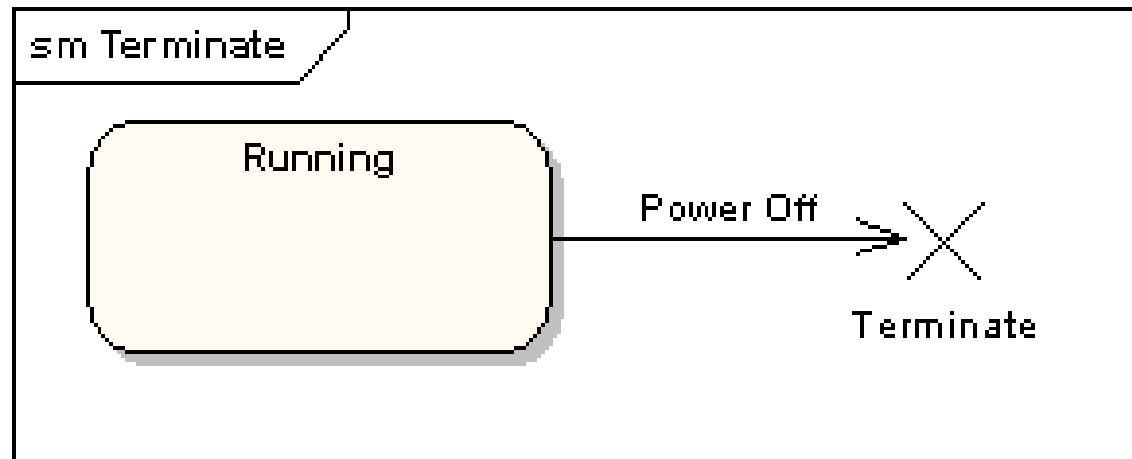
# Choice Pseudo State



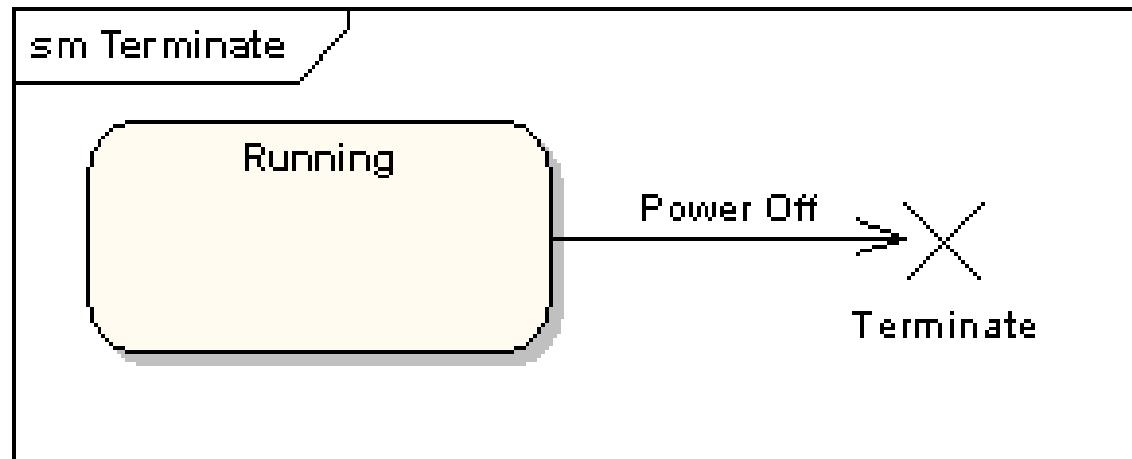
# Junction Pseudo State



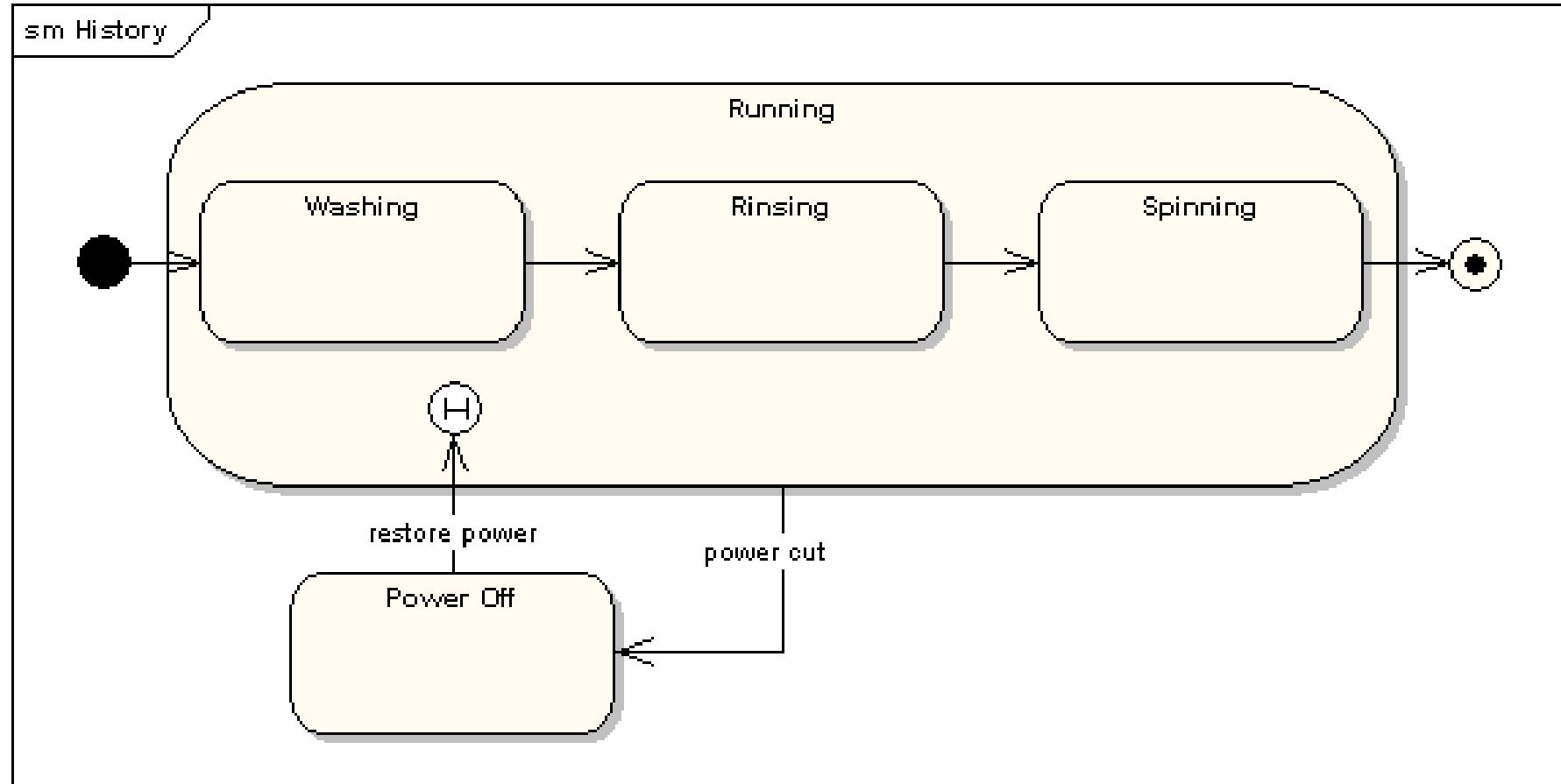
# Terminate Pseudo State



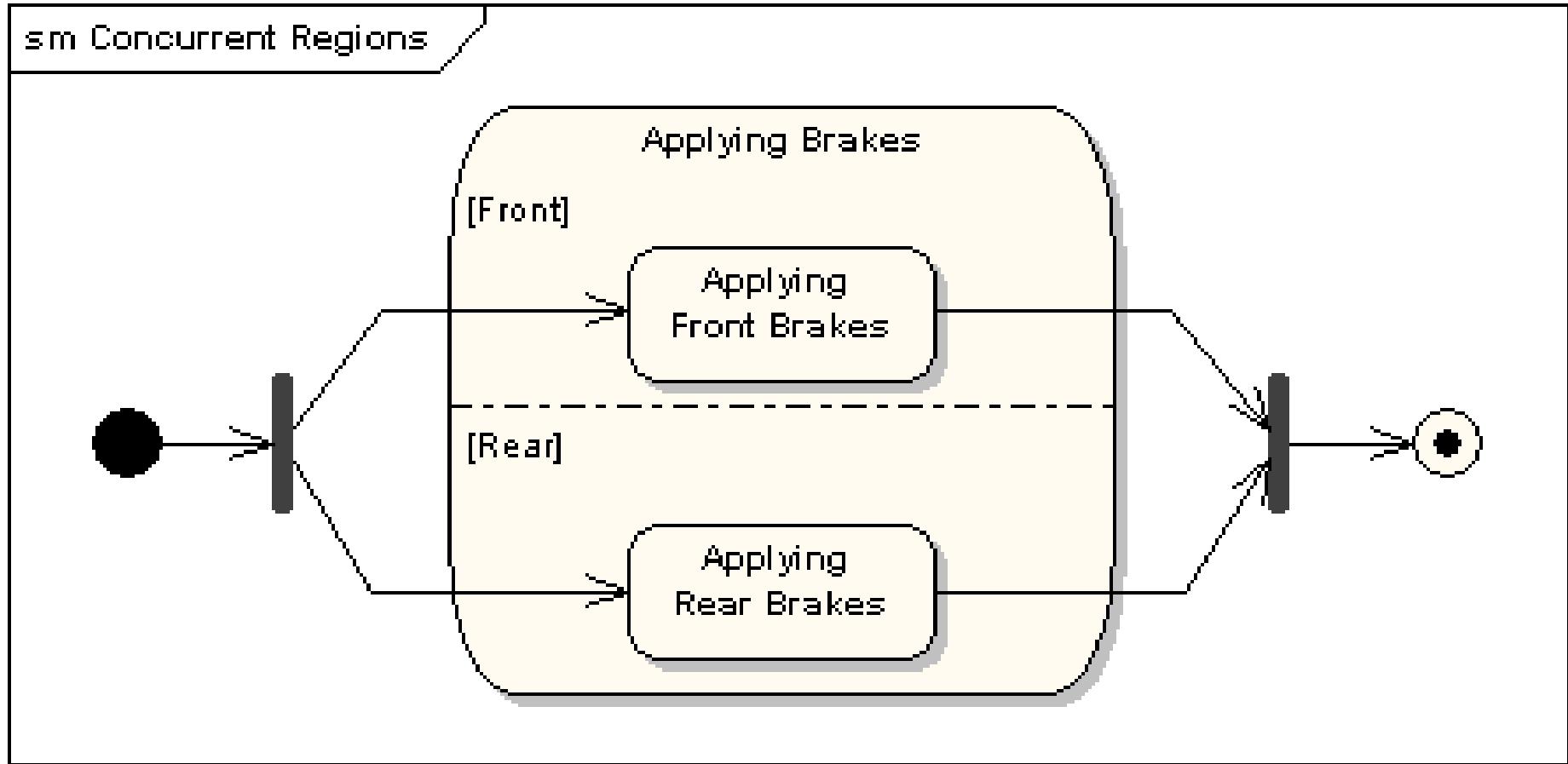
# Terminate Pseudo State



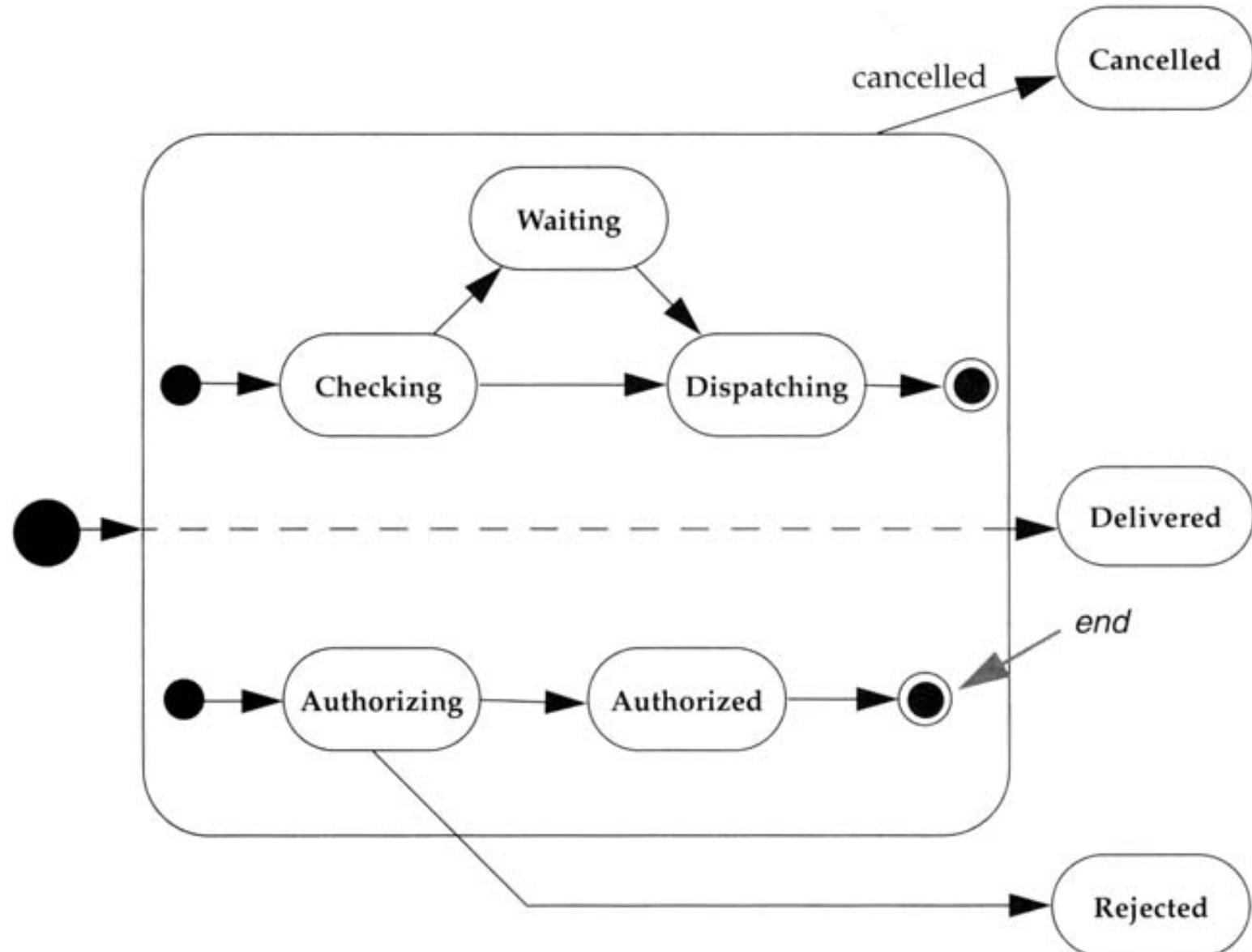
# History State



# Concurrent Regions within a State



# Concurrency in State Diagram



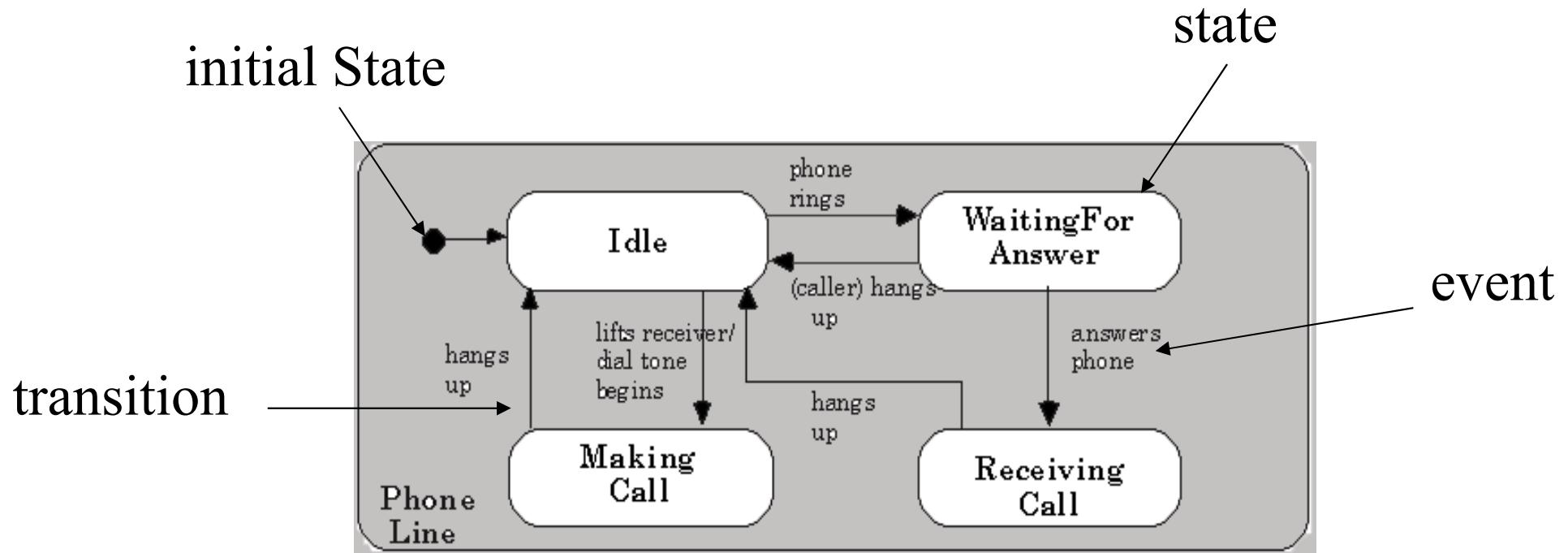
# UML: Statechart diagrams

- Represent behavior of single objects with interesting dynamic behavior in terms of states and transitions
- The behavior of the single object Watch, for example, has several different interesting states, BlinkHours, BlinkMinutes, BlinkSeconds,, etc.
- Because in each state pressing a button or two yields a different result.

# State Machine Diagrams

A State chart diagram shows the lifecycle of an object

- A *state* is a condition of an object for a particular time
- An *event* causes a *transition* from one state to another state
- Here is a State chart for a Phone Line object:

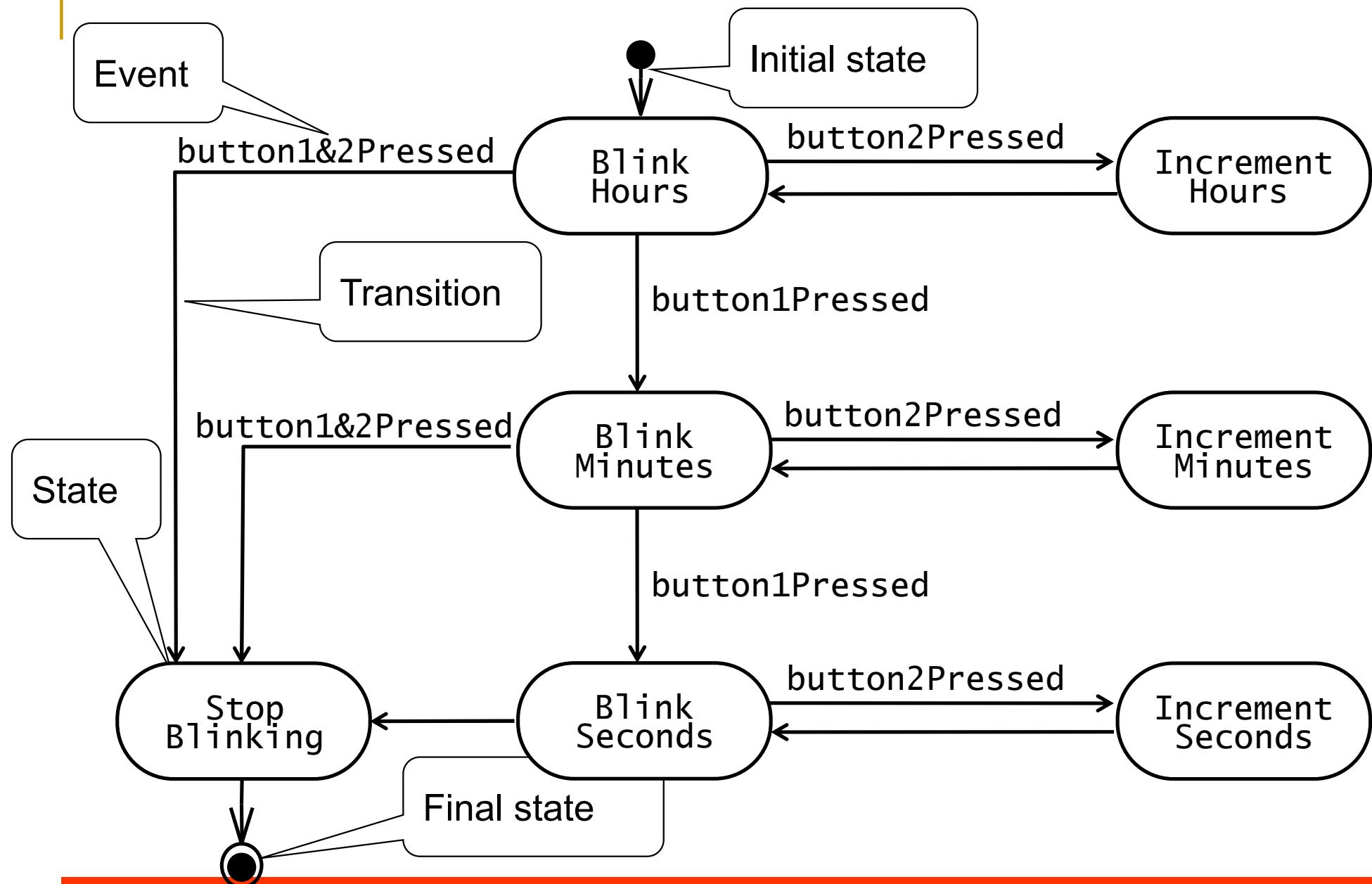


# When to develop a state chart?

Model objects that have change state in interesting ways:

- Devices (microwave oven, Ipod)
- Complex user interfaces (e.g., menus)
- Transactions (databases, banks, etc.)
- Stateful sessions (server-side objects)
- Role mutators (what role is an object playing?)
- Etc.

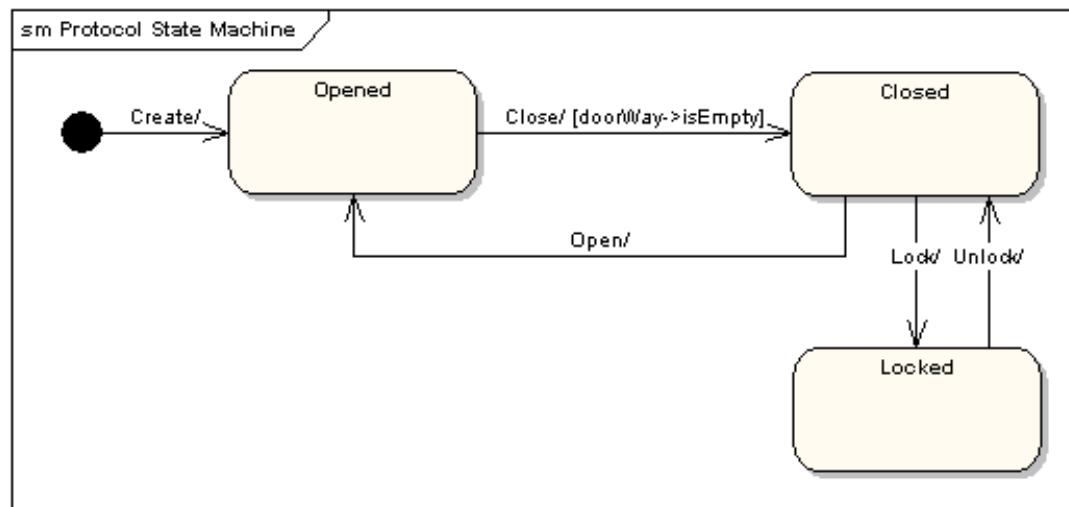
# UML: Statechart diagrams



Represent behavior of a *single object* with interesting dynamic behavior.

# State Machine Diagrams

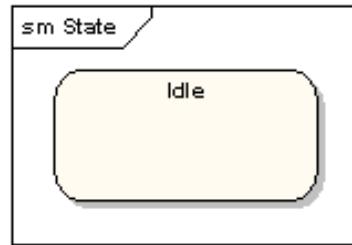
- A state machine diagram models the behavior of a single object, specifying the sequence of events that an object goes through during its lifetime in response to events. As an example, the following state machine diagram shows the states that a door goes through during its lifetime.
- The door can be in one of **three states**: "Opened", "Closed" or "Locked". It can respond to the events **Open**, **Close**, **Lock** and **Unlock**. Notice that **not all events are valid in all states**; for example, if a door is opened, you cannot lock it until you close it. Also notice that a state transition can have a guard condition attached: if the door is Opened, it can only respond to the Close event if the condition



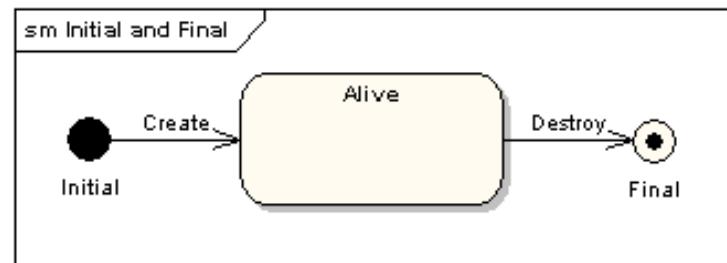
`doorWay->isEmpty` is fulfilled. The syntax and conventions used in state machine diagrams will be discussed in full in the following sections.

# State Machine Diagrams

- **States** - A state is denoted by a round-cornered rectangle with the name of the state written inside it.

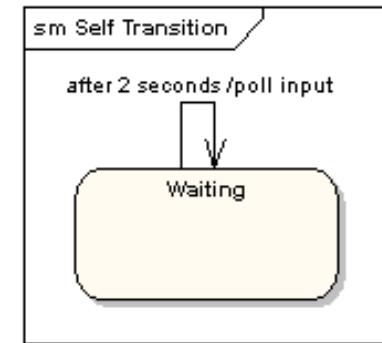
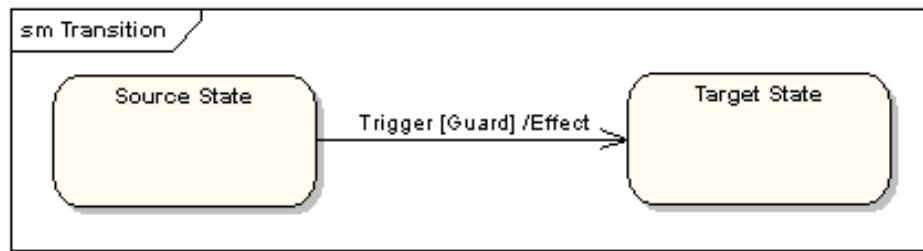


- **Initial and Final States** - The initial state is denoted by a filled black circle and may be labeled with a name. The final state is denoted by a circle with a dot inside and may also be labeled with a name.



# State Machine Diagrams

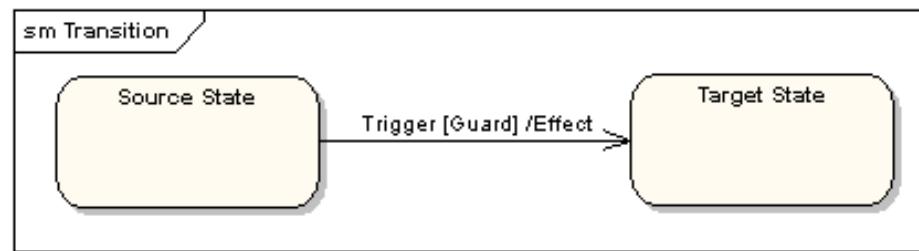
- ***Transitions*** - Transitions from one state to the next are denoted by lines with arrowheads. A transition may have a trigger, a guard and an effect, as below.
- ***Self-Transitions*** - A state can have a transition that returns to itself, as in the following diagram. This is most useful when an effect is associated with the transition.



- "***Trigger***" is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time. "***Guard***" is a condition which must be true in order for the trigger to cause the transition. "***Effect***" is an action which will be invoked directly on the object that owns the state machine as a result of the transition.

# State Machine Diagrams

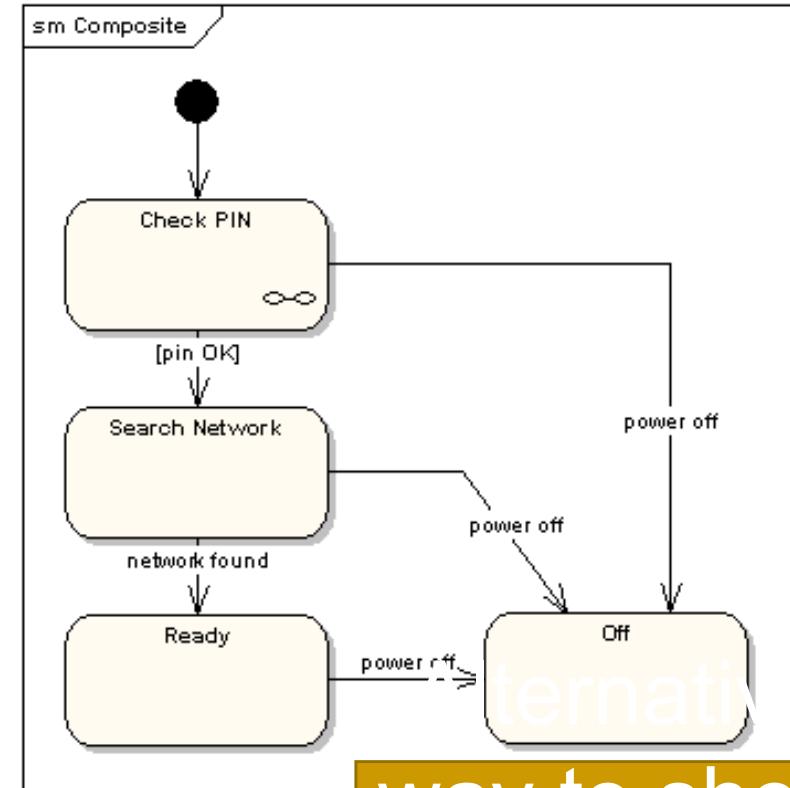
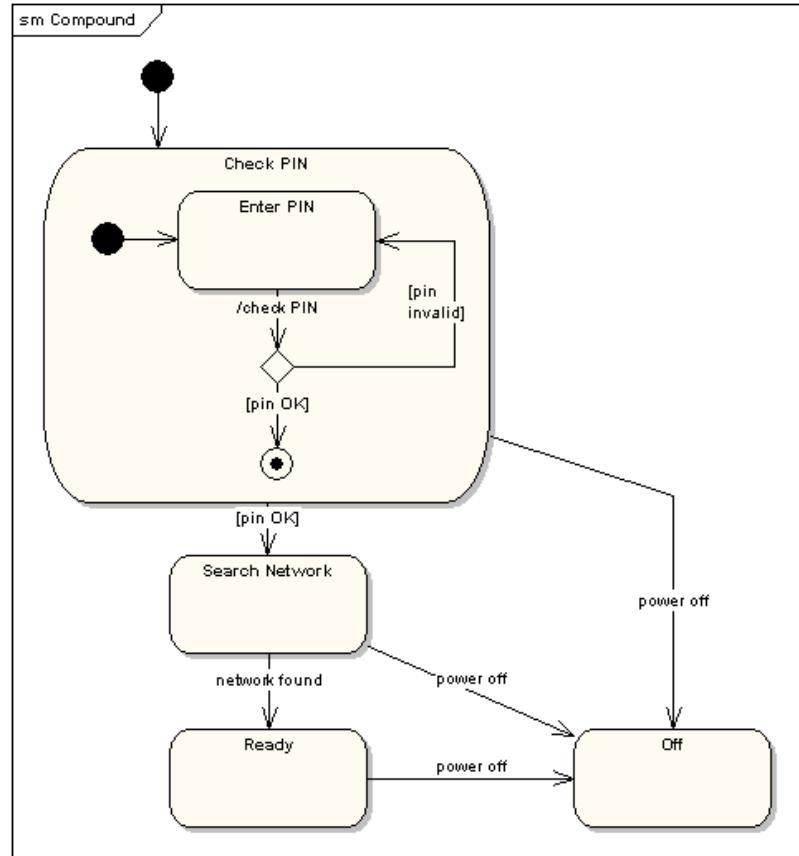
- ***State Actions*** - In the transition example above, an effect was associated with the transition. If the target state had many transitions arriving at it, and each transition had the same effect associated with it, it would be better to associate the effect with the target state rather than the transitions. This can be done by defining an entry action for the state. The diagram below shows a state with an entry action and an exit action.



- It is also possible to define actions that occur on events, or actions that always occur. It is possible to define any number of actions of each type.

# State Machine Diagrams

- **Compound States** - A state machine diagram may include sub-machine diagrams, as in the example below.

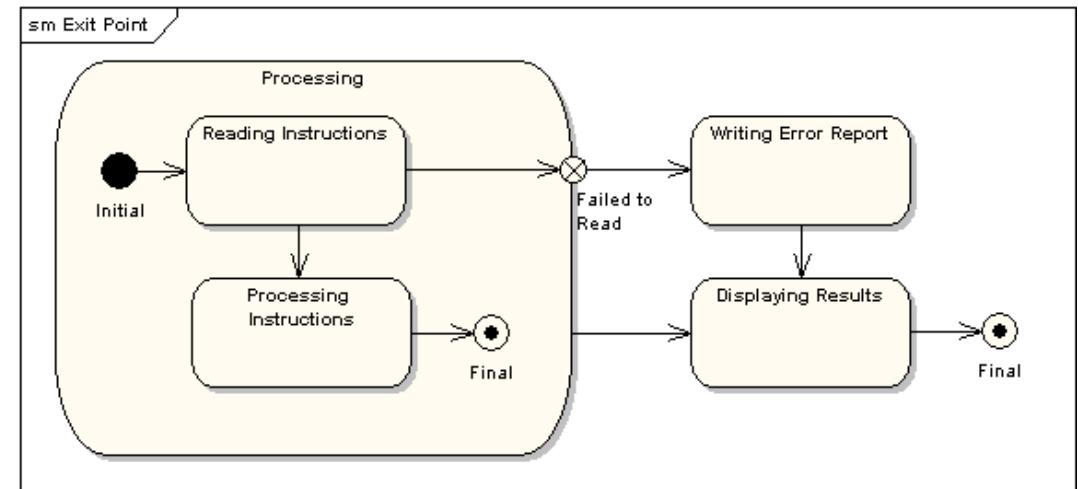
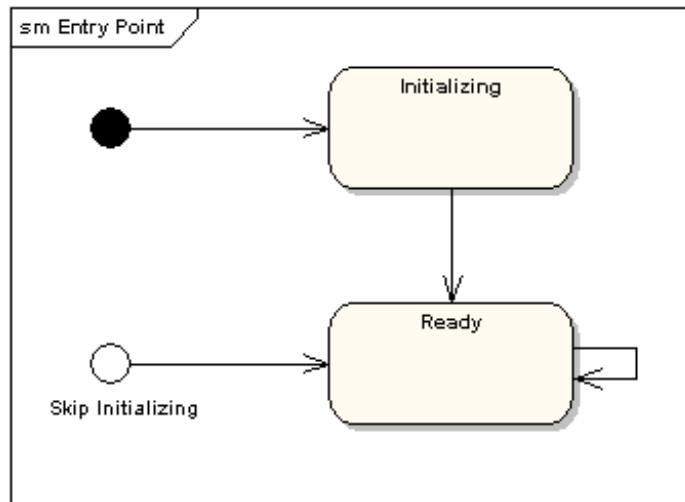


way to show  
the same

- The  $\infty$  symbol indicates that details of the Check PIN sub-machine are shown in a separate diagram.

# State Machine Diagrams

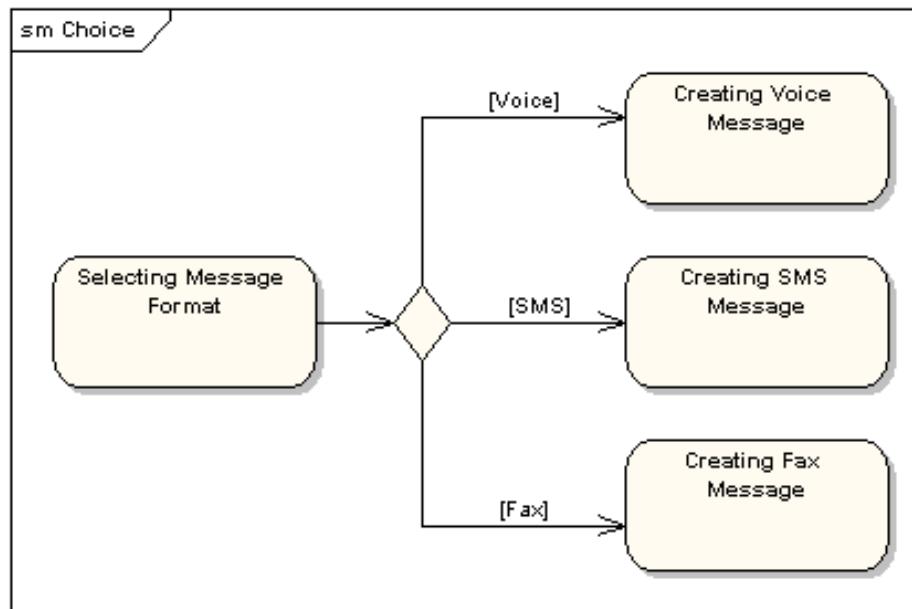
- **Entry Point** - Sometimes you won't want to enter a sub-machine at the normal initial state. For example, in the following sub-machine it would be normal to begin in the "Initializing" state, but if for some reason it wasn't necessary to perform the initialization, it would be possible to begin in the "Ready" state by transitioning to the named entry point.



- **Exit Point** - In a similar manner to entry points, it is possible to have named alternative exit points. The following diagram gives an example where the state executed after the main processing state depends on which route is used to transition out of the state.

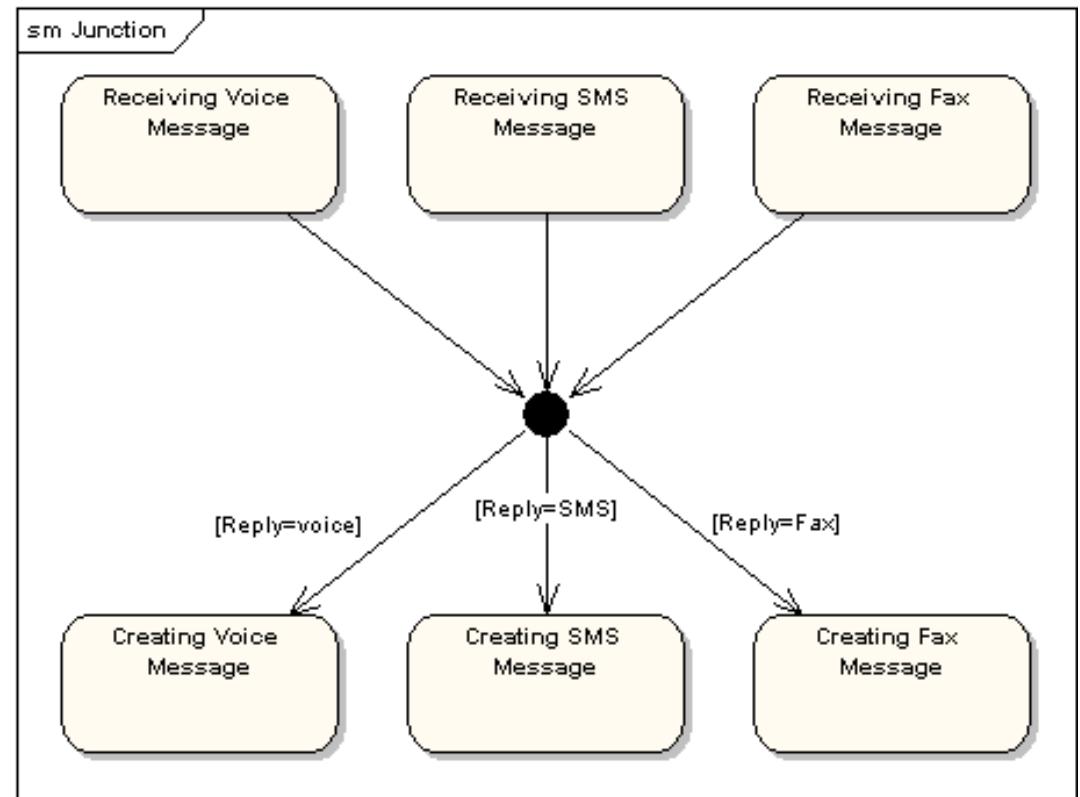
# State Machine Diagrams

- **Choice Pseudo-State** - A choice pseudo-state is shown as a diamond with one transition arriving and two or more transitions leaving. The following diagram shows that whichever state is arrived at, after the choice pseudo-state, is dependent on the message format selected during execution of the previous state.



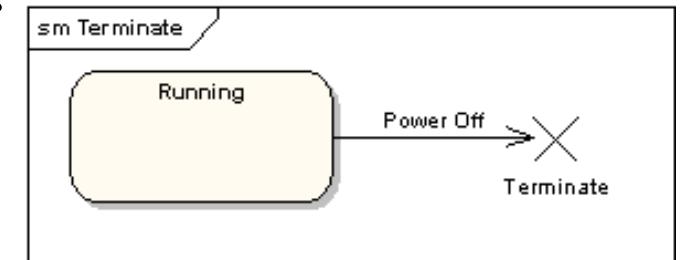
# State Machine Diagrams

- **Junction Pseudo-State** - Junction pseudo-states are used to chain together multiple transitions. A **single junction can have one or more incoming, and one or more outgoing, transitions**; a guard can be applied to each transition.
- A junction which splits an incoming transition into multiple outgoing transitions **realizes a static conditional branch**, as opposed to a choice pseudo-state which realizes a dynamic conditional branch.

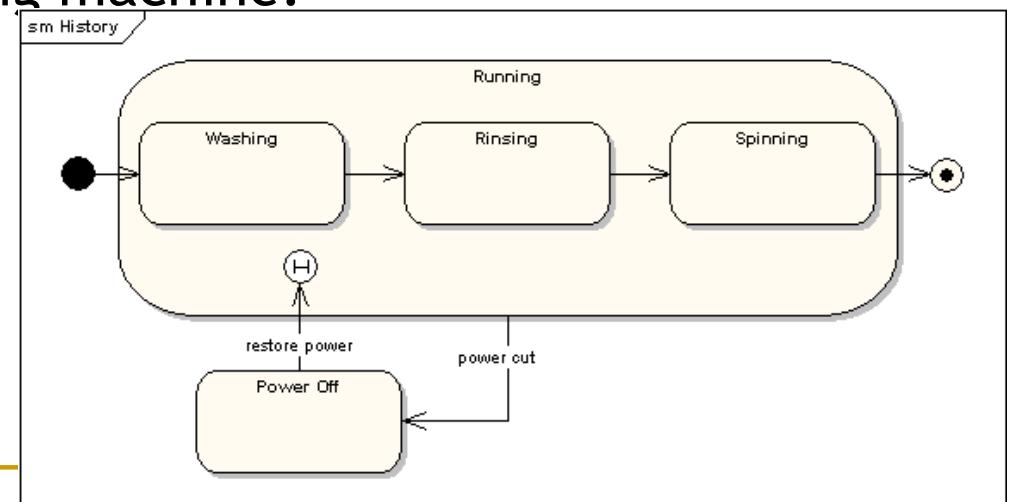


# State Machine Diagrams

- **Terminate Pseudo-State** - Entering a terminate pseudo-state indicates that the lifeline of the state machine has ended. A terminate pseudo-state is notated as a cross.

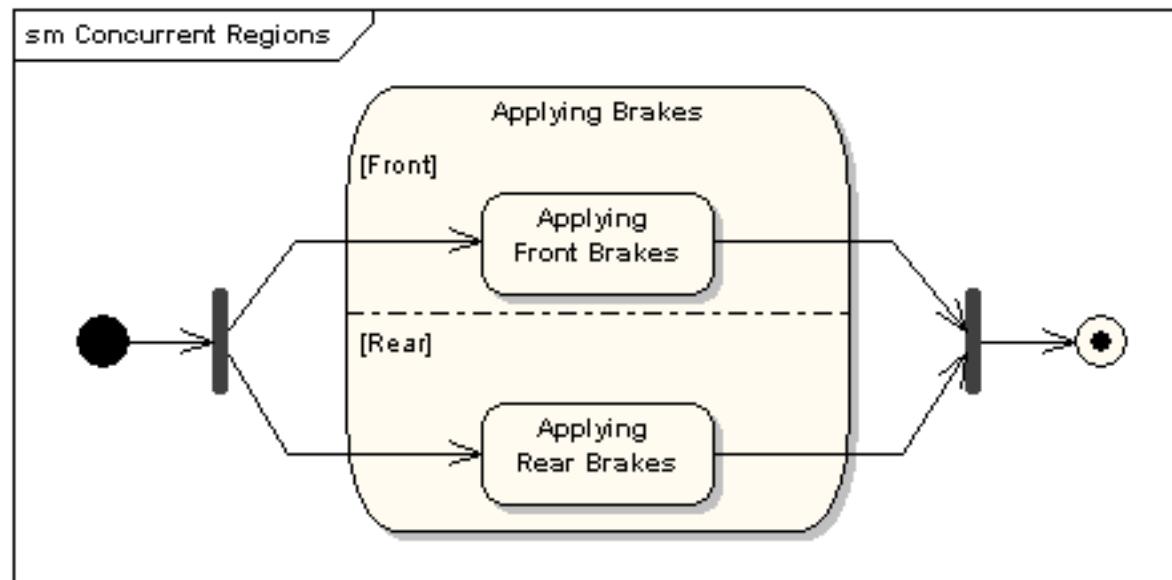


- **History States** - A history state is used to remember the previous state of a state machine when it was interrupted. The following diagram illustrates the use of history states. The example is a state machine belonging to a washing machine.



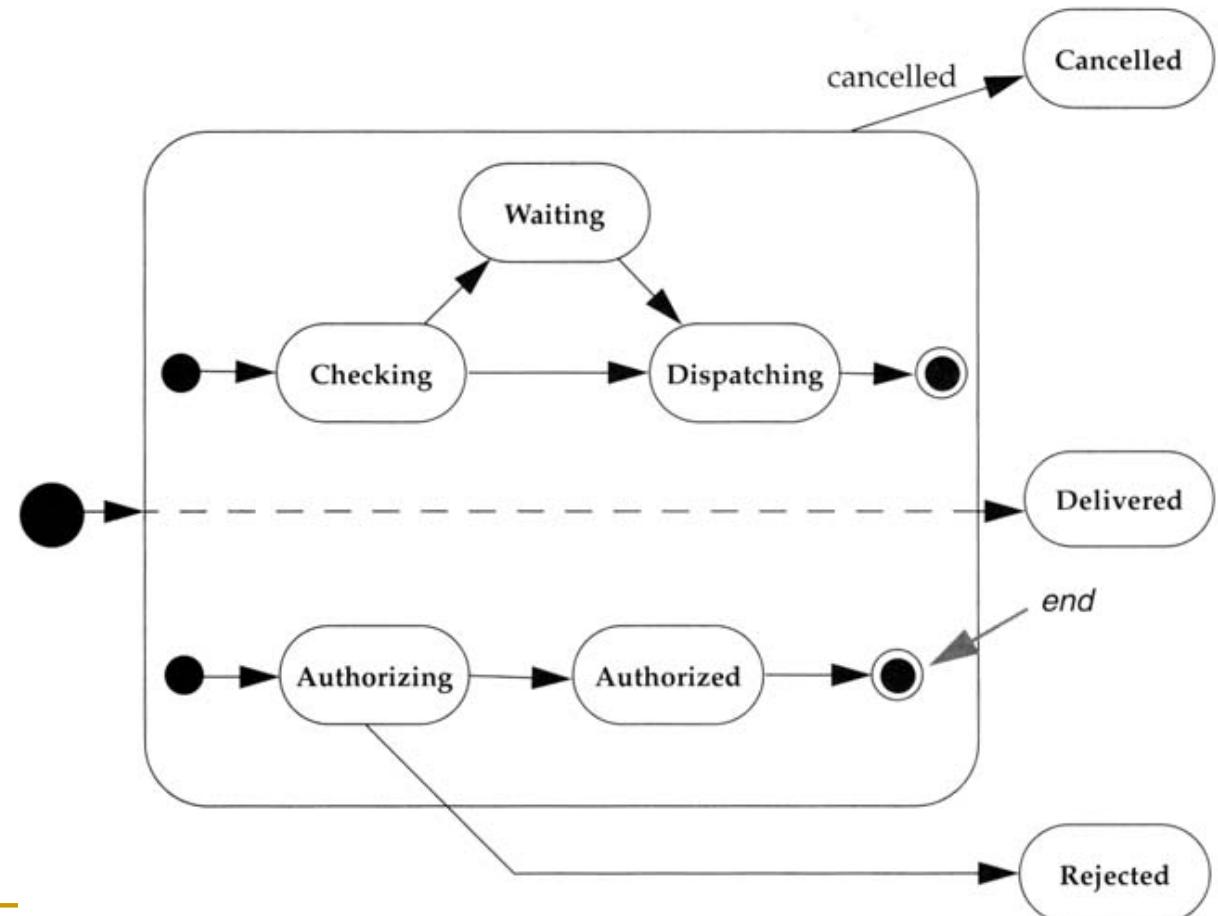
# State Machine Diagrams

- **Concurrent Regions** - A state may be divided into regions containing sub-states that exist and execute concurrently. The example below shows that within the state "Applying Brakes", the front and rear brakes will be operating simultaneously and independently. Notice the use of fork and join pseudo-states, rather than choice and merge pseudo-states. These symbols are used to synchronize the concurrent threads.

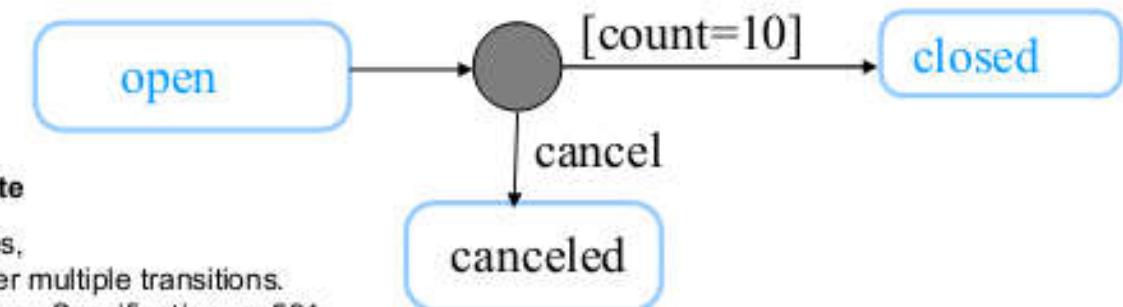
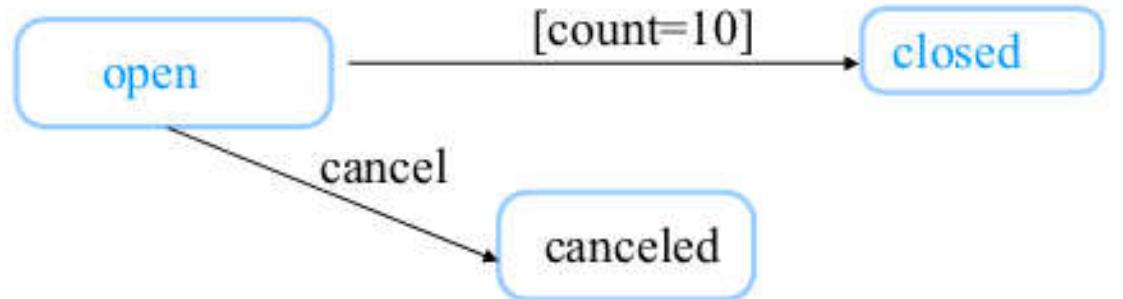


# Concurrency in state diagrams

- Dashed line indicates that an order is in two different states, e.g. Checking & Authorizing
- When order leaves concurrent states, it's in a single state: Canceled, Delivered or Rejected

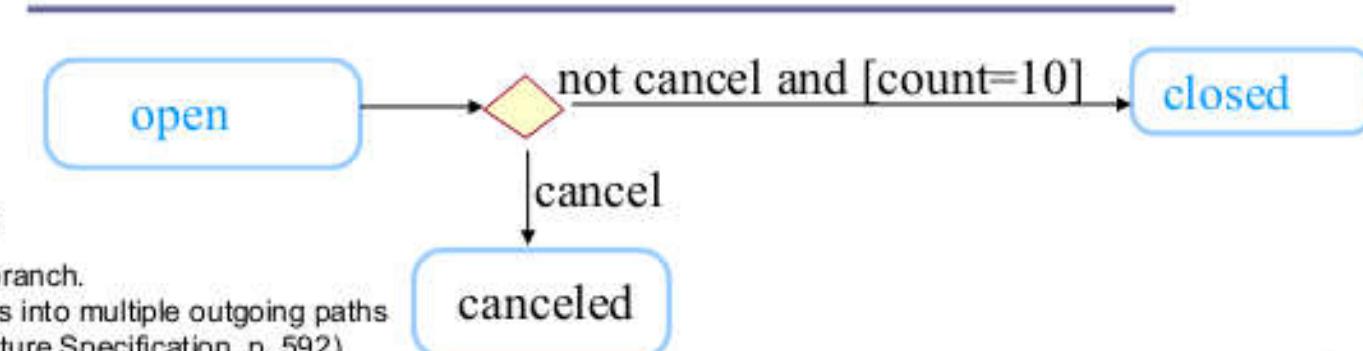


# State Transitions – notational variation



## Junction pseudo state

- semantic-free vertices,
- used to chain together multiple transitions.
- UML 2.0 Superstructure Specification, p. 591



## Choice pseudo state

- dynamic conditional branch.
- splitting of transitions into multiple outgoing paths
- UML 2.0 Superstructure Specification, p. 592

Thank You