Object-Oriented Programming

LECTURE#07-15

Dr. Sanjeev Patel

Asst. Professor,

Department of Computer Science and Engineering

National Institute of Technology Rourkela, Odisha

Outline

- Objects and Class
- Class Attributes
- Object Behavior
- Object Methods
- Characteristics of Object-oriented systems
- Abstraction and Encapsulation
- Information Hiding
- Inheritance or Class Hierarchy
- Polymorphism
- Exception handling
- Object containers or Containment
- Object identity
- Persistence

Object

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

Class

- Class is a definition, or description, of how the object is supposed to be created
  - what it contains and how it work
- The object instance is a composite data type, or object, created based on the rules set forth in the class definition.
- The point of object-based programming languages is that
  - they give the user the ability to define their own data types that can be specifically required to the application

Objects and Object classes

- Objects are entities in a software system which represent instances of real-world and system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

Objects and Object Classes

✓An **object** is an entity that has a state and a defined set of operations which operate on that state.

✓The state is represented as a set of object attributes.

✓The operations associated with the object provide services to other objects (clients) which request these services when some computation is required.

✓Objects are created according to some **object class** definition.

✓An object class definition serves as a template for objects.

✓It includes declarations of all the attributes and services which should be associated with an object of that class.

OBJECT STATE & BEHAVIOUR

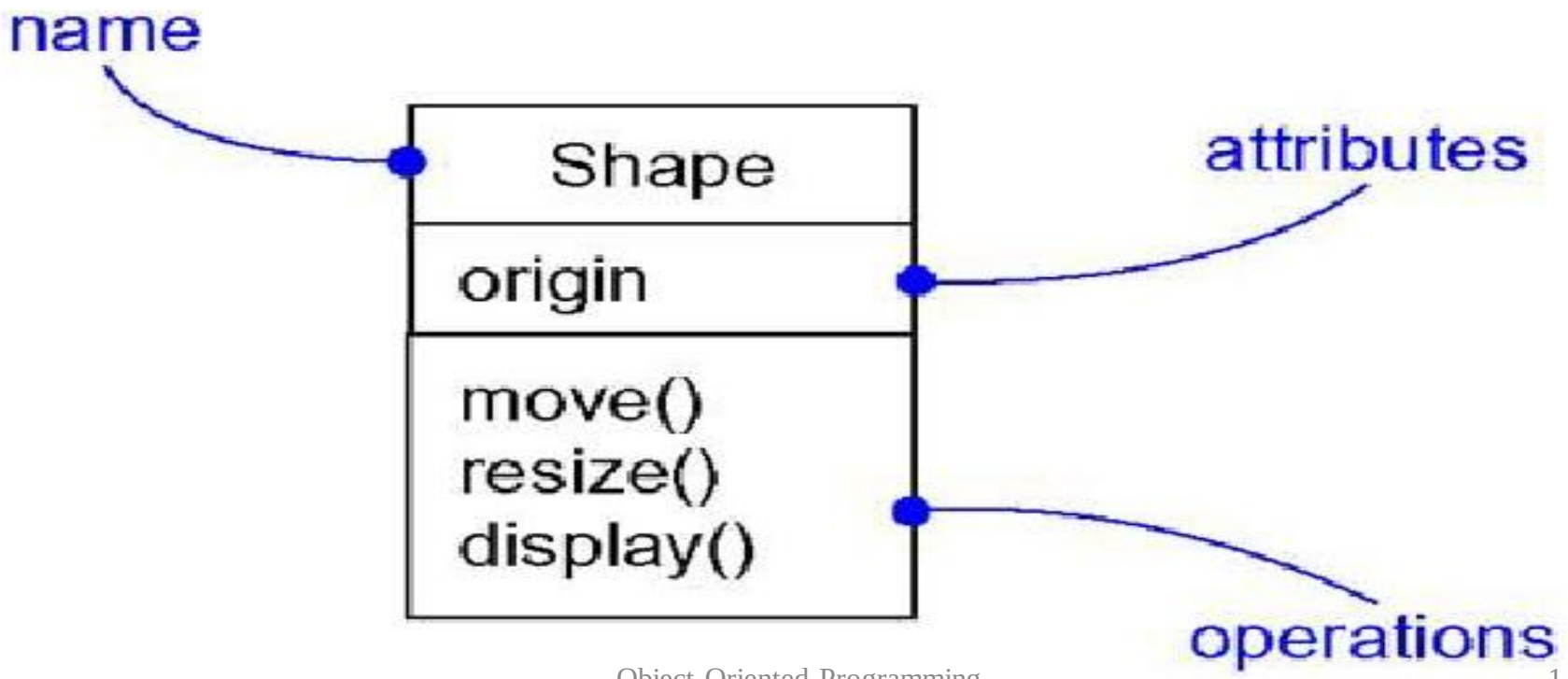
- Real-world objects share two characteristics
- They all have state and behavior.
- State
  - Every object, at any given point of time would have to have a set of attributes defining its State.
- Behavior
  - Every object based on its state and optionally identity will have particular behavior.
- E.g. Bicycles have
  - state (current gear, current pedal cadence, current speed) and
  - behavior (changing gear, changing pedal cadence, applying brakes)

THE PROPERTY (OBJECT ATTRIBUTE)

- Properties are variables contained in the class
- Every instance of the object has those properties.
- Properties should be set in the prototype property of the class (function) so that inheritance works correctly.
- Methods follow the same logic as properties
- The difference is that they are functions and they are defined as functions

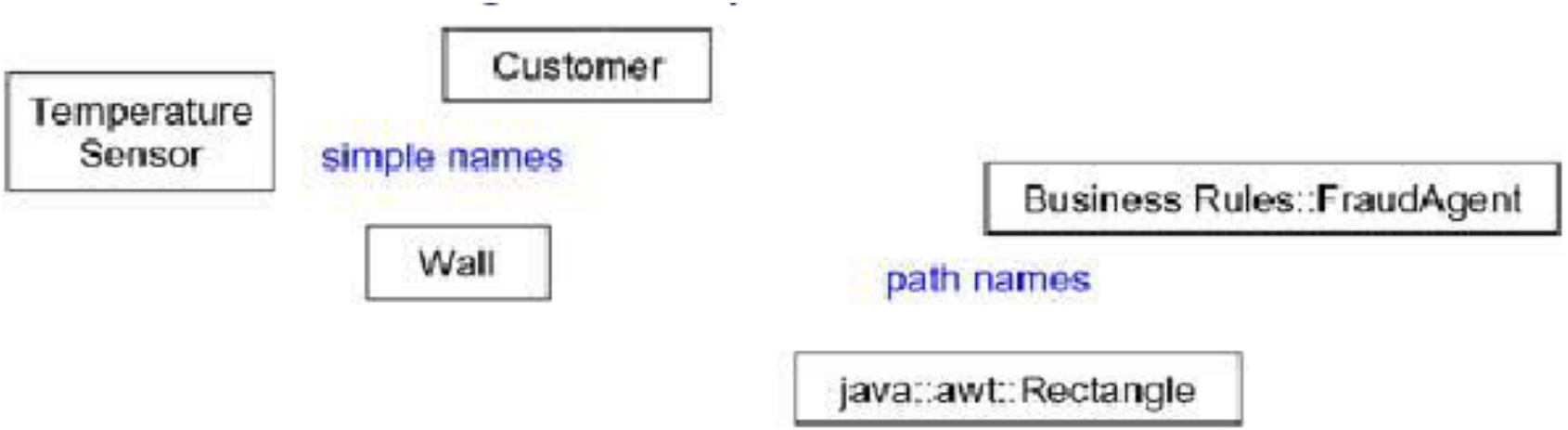
Class

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



Class

- Every class must have a name that distinguishes it from other classes.
- A *name is a textual string.*
- That name alone is known as a *simple name; a path name is the class name prefixed by the* name of the package in which that class lives.



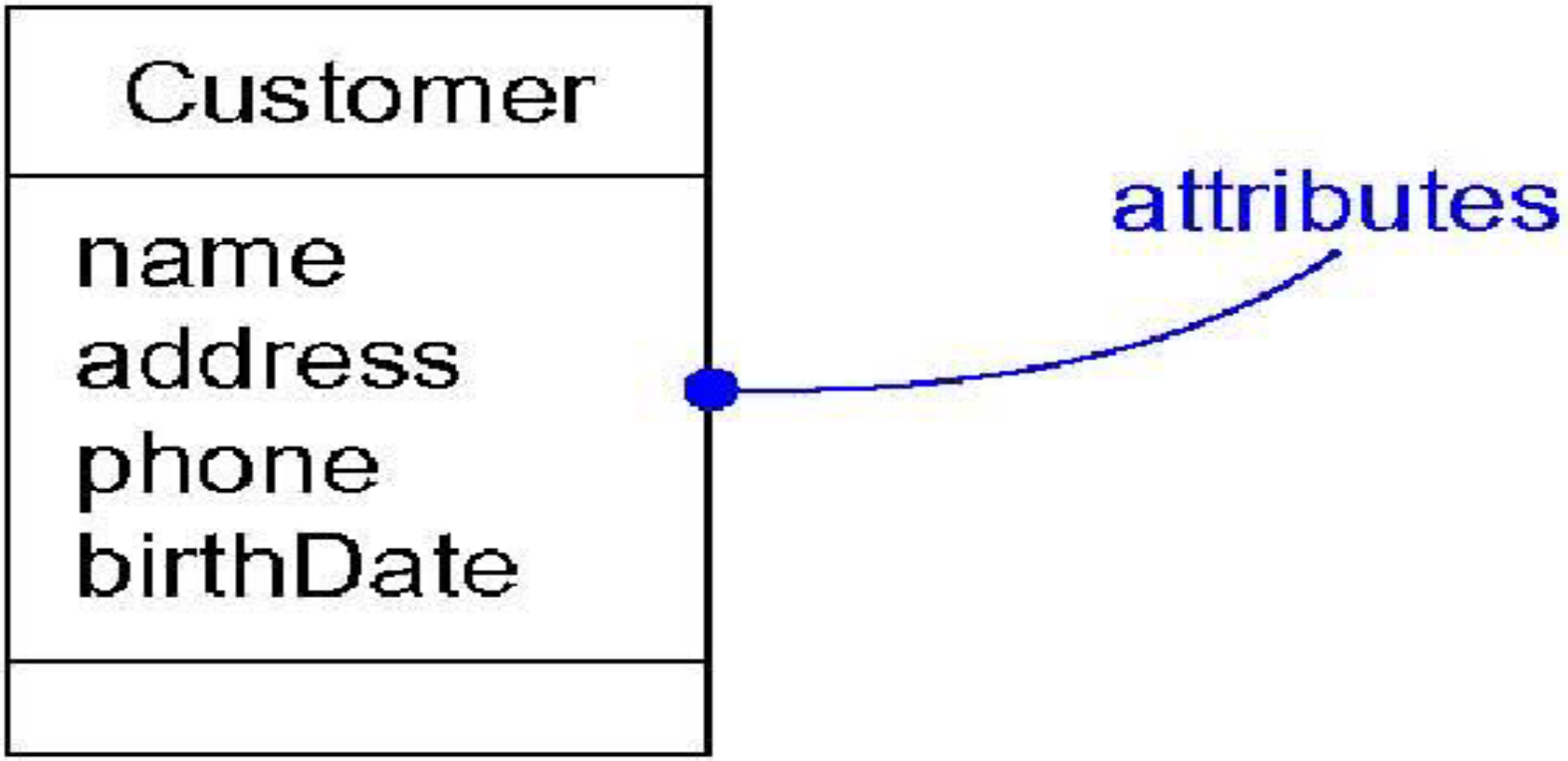
Class

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

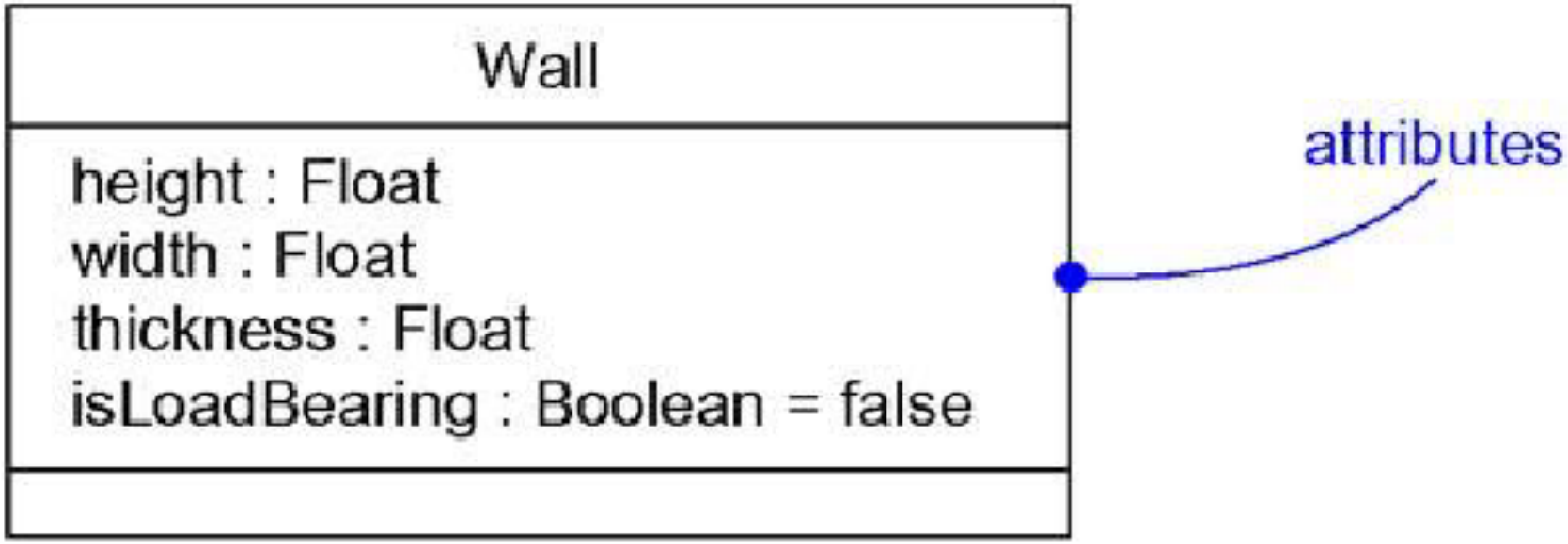
Attributes

- An *attribute is a named property of a class that describes a range of values that instances of the* property may hold.
- An attribute represents some property of the thing you are modeling that is shared by all objects of that class.
- For example, every wall has a height, width, and thickness.
- You might model your customers in such a way that each has a name, address, phone number, and date of birth.
- An attribute is therefore an abstraction of the kind of data or state an object of the class might encompass.

Attributes



Attributes and Their Class

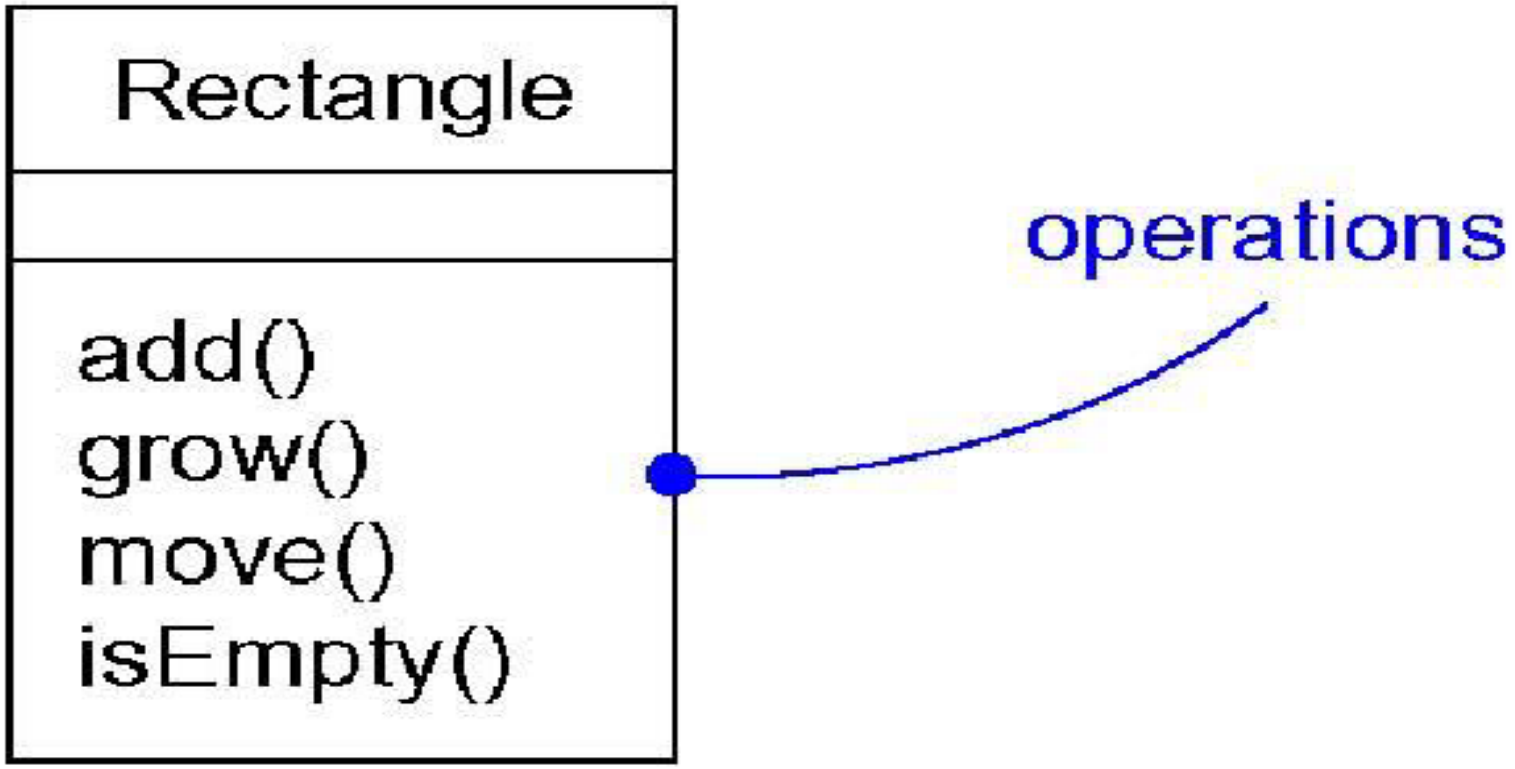


You can specify an attribute by stating its class and possibly a default initial value,

Operations

- An *operation is the implementation of a service that can be requested from any object of the class* to affect behavior.
- In other words, an operation is an abstraction of something you can do to an object and that is shared by all objects of that class.
- For example, in a windowing library such as the one found in Java's awt package, all objects of the class Rectangle can be moved, resized, or queried for their properties.

Operations



Object Communication

- Conceptually, objects communicate with the help of message passing.
- Messages
  - The name of the service requested by the calling object;
  - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
  - Name = procedure name;
  - Information = parameter list.

Message examples

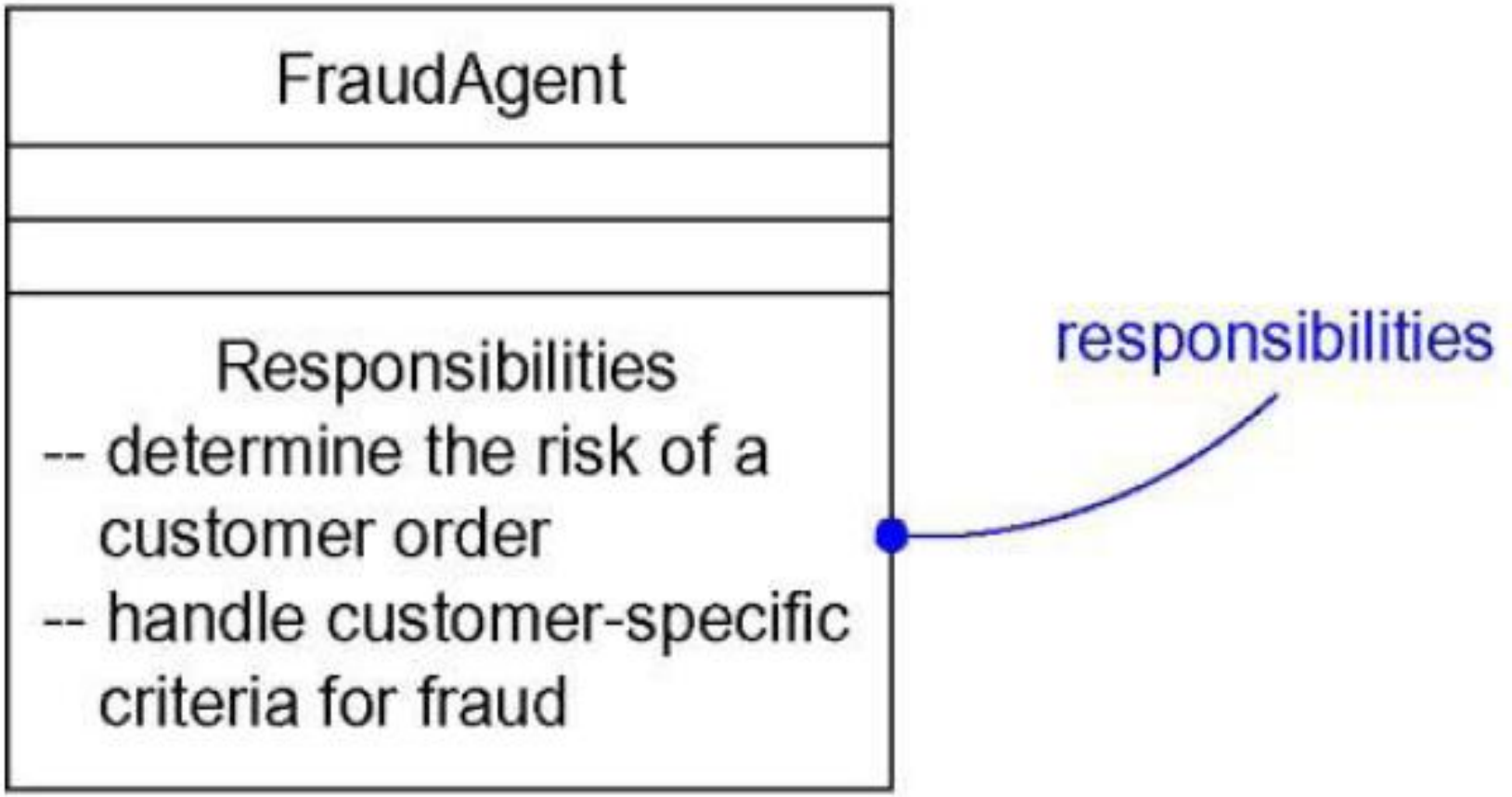
```
// Call a method associated with a buffer
//  object that returns the next value
//   in the buffer
        v = circularBuffer.Get () ;

// Call the method associated with a
//  thermostat object that sets the
// temperature to be maintained
        thermostat.setTemp (20) ;
```

Responsibilities

- A *responsibility is a contract or an obligation of a class.*
  - A Wall class is responsible for knowing about height, width, and thickness.
  - A FraudAgent class, as you might find in a credit card application, is responsible for processing orders and determining if they are legitimate, suspect, or fraudulent.
  - A TemperatureSensor class is responsible for measuring temperature and raising an alarm if the temperature reaches a certain point.

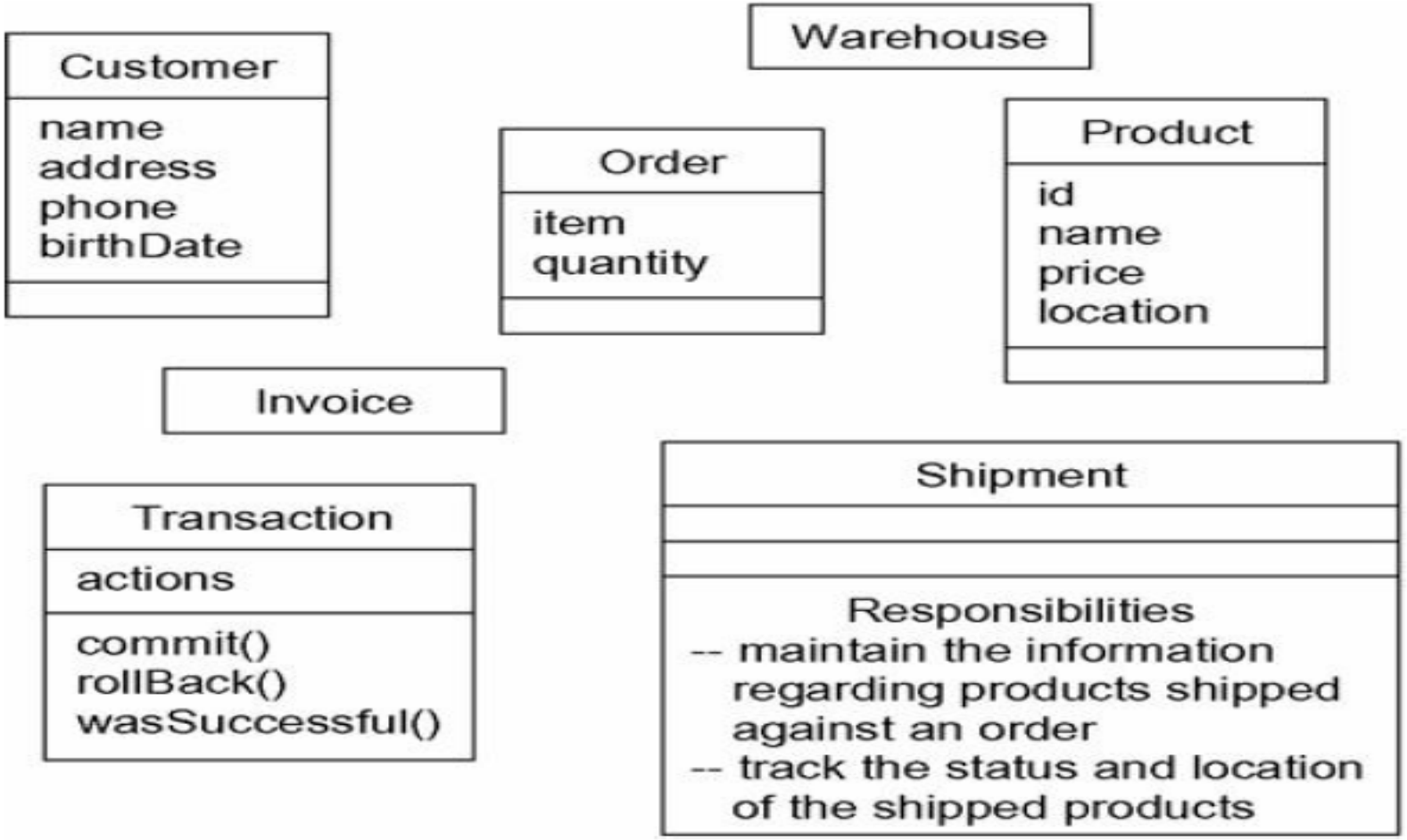
Responsibilities



To Model The Vocabulary of A System

- Identify those things that users or implementers use to describe the problem or solution.
- Use CRC cards and use case-based analysis to help find these abstractions.
- For each abstraction, identify a set of responsibilities.
- Provide the attributes and operations that are needed to carry out these responsibilities for each class.

Modeling the Vocabulary of a System



- A set of classes drawn from a retail system

Modeling the Vocabulary of a System

- A set of classes drawn from a retail system, including Customer, Order, and Product.
- A few other related abstractions drawn from the vocabulary of the problem, such as
  - Shipment (used to track orders),
  - Invoice (used to bill orders), and
  - Warehouse (where products are located prior to shipment).
- There is also one solution-related abstraction, Transaction, which applies to orders and shipments.

Object-Oriented Programming

- Object-oriented programming is a method of implementation in which
- Programs are organized as cooperative collections of objects
  - each of which represents an instance of some class,
  - and whose classes are all members of a hierarchy of classes united via inheritance relationships.

Object-Oriented Programming

- Object-oriented programming uses objects, not algorithms,
  - as its fundamental logical building blocks
- Each object is an instance of some class
- Classes may be related to one another via inheritance relationships

Characteristics of Object-oriented systems

- The Elements of the Object Model
  - Abstraction
  - Encapsulation
  - Modularity
  - Hierarchy/Inheritance
- The other characteristics are
  - Data hiding
  - Polymorphism/Typing
  - Concurrency
  - Persistence

Abstraction

- Abstraction is one of the fundamental ways that we as humans cope with complexity.
- A simplified description, or specification, of a system that emphasizes some of the system's details or properties while suppressing others.
- A good abstraction is one that emphasizes details that are significant to the reader or user
  - and suppresses details that are, at least for the moment, immaterial or diversionary

Abstraction

- An abstraction focuses on the outside view of an object and so serves to separate an object's essential behavior from its implementation
- There is a spectrum of abstraction, from objects which closely model problem domain entities to objects.
- There are different types of abstraction
- Entity abstraction
  - An object that represents a useful model of a problem domain or solution domain entity

Abstraction

- Action abstraction
  - An object that provides a generalized set of operations, all of which perform the same kind of function
- Virtual machine abstraction
  - An object that groups operations that are all used by some superior level of control, or operations that all use some junior-level set of operations
- Coincidental abstraction
  - An object that packages a set of operations that have no relation to each other

Abstraction

- Abstraction is the concept of object-oriented programming
  - That shows only essential attributes
  - and hides unnecessary information.
- The main purpose of abstraction is hiding the unnecessary details from the users.
- Abstraction is selecting data from a larger pool to
  - show only relevant details of the object to the user.
  - It helps in reducing programming complexity and efforts

Example of Abstraction

Abstraction: Temperature Sensor
Important Characteristics: temperature location
Responsibilities: report current temperature calibrate

Abstraction Hierarchy

- For complex problems
  - A single level of abstraction is inadequate
- A hierarchy of abstractions needs to be constructed
- A model in a layer is abstraction of the lower layer models
- Higher layer models implement the model in the layer

Encapsulation

- Simply stated, the abstraction of an object should precede the decisions about its implementation.
- Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.
- Abstraction and encapsulation are complementary concepts

Encapsulation

- Abstraction focuses on the observable behavior of an object,
  - whereas encapsulation focuses on the implementation that gives rise to this behavior.
- Encapsulation is most often achieved through information hiding (not just data hiding),
  - which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics
- Encapsulation provides explicit barriers among different abstractions and thus leads to a clear separation of concerns

Encapsulation

- For example, consider again the structure of a plant.
- To understand how photosynthesis works at a high level of abstraction
  - we can ignore details such as the responsibilities of plant roots or
  - the chemistry of cell walls.
- Similarly, in designing a database application, it is standard practice to write programs so that
  - they don't care about the physical representation of data but
  - depend only on a schema that denotes the data's logical view .
- In both of these cases, objects at one level of abstraction are shielded from implementation details at lower levels of abstraction

Encapsulation

- For abstraction to work, implementations must be encapsulated according to Liskov's view
- This means that each class must have two parts an interface and an implementation.
  - The interface of a class captures only its outside view, encompassing our abstraction of the behavior common to all instances of the class.
  - The implementation of a class comprises the representation of the abstraction as well as the mechanisms that achieve the desired behavior.
  - The interface of a class is the one place where we assert all of the assumptions
  - that a client may make about any instances of the class

Encapsulation [6]

- Encapsulation is the mechanism that binds together code and the data it manipulates
- It keeps both safe from outside interference and misuse.
- In an object-oriented language, code and data may be combined in such a way that a self-contained "black box" is created.
- An object is the device that supports encapsulation. Within an object, code, data, or both may be private to that object or public.

Encapsulation [5, 6]

- Private code or data is known to and accessible only by another part of the object
- Private data may not be accessed by a piece of the program that exists outside the object.
- When code or data is public, other parts of your program may access it even though it is defined within an object.
- https://www.edureka.co/blog/data-hiding-in-cpp/

Example of Encapsulation [5]

```
class Encapsulation {
   private:
     int num; // data hidden from outside
    world
    public:
     // function to set value of  variable x
     void set(int a)
     {     num =a;    }
     // function to return value of
     // variable x
     int get()
     {  return num;  }
};
```

```
int main()
{
     Encapsulation obj;
      obj.set(5);
      cout<<obj.get();
      return 0;
}
```

Output: 5

Abstraction and Data Hiding [5]

- **Data Abstraction**
  - It is a mechanism of hiding the implementation from the user & exposing the interface.
- **Data Hiding**
  - It is a process of combining data and functions into a single unit.
  - The aim of data hiding is to conceal data within a class, to prevent its direct access from outside the class.
  - It helps programmers to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes.
- **Data Hiding vs Data Encapsulation**
  - data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods.

Example of Abstraction[5]

```
class Abstraction
{
   private:
      int num1, num2;
   public:
      void set(int a, int b)
      {
         num1 = a;   num2 = b;
      }
void display()
      {
         cout<<"num1 = " <<num1 << endl;
         cout<<"num2 = " << num2 << endl;
      }
};
```

```
int main()
{
   Abstraction obj;
   obj.set(50, 100);
   obj.display();
   return 0;
}
```

Output: num1=50
num2=100

Example of Data Hiding [5]

```
class Base{
   int num;  //by default private
   public:
   void read();
   void print();
};
void Base :: read(){
   cout<<"Enter any Integer value"<<endl;
    cin>>num;
}
  void Base :: print(){
   cout<<"The value is "<<num<<endl;
}
```

```
int main() {
   Base obj;
   obj.read();
   obj.print();
   return 0;
}
```

Output: Enter any Integer value

10

The value is 10

Modularity

- The act of partitioning a program into individual components can reduce its complexity to some degree.
- Modularization consists of dividing a program into modules
  - which can be compiled separately, but which have connections with other modules.
- The connections between modules are the assumptions which the modules make about each other
- Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules

Modularity: Example

- Consider the Example of Hydroponics Gardening System.
- Suppose we decide to use a commercially available workstation where the user can control the system's operation.
- At this workstation, an operator could create new growing plans, modify old ones, and follow the progress of currently active ones.
- Since one of our key abstractions here is that of a growing plan,
  - we might therefore create a module
  - whose purpose is to collect all of the classes associated with individual growing plans (e.g., FruitGrowingPlan, GrainGrowingPlan).
- The implementations of these GrowingPlan classes would appear in the implementation of this module.
- We might also define a module whose purpose is to collect all of the code associated with all user interface functions.

The Meaning of Hierarchy

- Hierarchy is a ranking or ordering of abstractions
- The two most important hierarchies in a complex system are
  - its class structure (the “is a” hierarchy) and
  - its object structure (the “part of” hierarchy)
- Inheritance is the most important “is a” hierarchy
  - Semantically, inheritance denotes an “is a” relationship
- Inheritance thus represents a hierarchy of abstractions, in which a subclass inherits from one or more super-classes

Meaning of Typing

- The concepts of strong and weak typing and static and dynamic typing are entirely different.
- Strong and weak typing refers to type consistency, whereas static and dynamic typing
  - refers to the time when names are bound to types.
- Static typing also known as static binding or early binding
  - means that the types of all variables and expressions are fixed at the time of compilation

Meaning of Typing

- Dynamic typing also known as late binding
  - the types of all variables and expressions are not known until runtime
- Polymorphism is a condition that exists when the features of dynamic typing and inheritance interact.
- Polymorphism represents a concept in type theory in which a single name (such as a variable declaration) may denote
  - objects of many different classes that are related by some common superclass.
  - Any object denoted by this name is therefore able to respond to some common set of operations

The Meaning of Concurrency

- The object is a concept that unifies these two different viewpoints
- Each object drawn from an abstraction of the real world may represent a separate thread of control a process abstraction.
- Such objects are called active
- Concurrency is the property that distinguishes an active object from one that is not active.

Concurrency

- Once concurrency is introduced into a system,
  - you must consider how active objects synchronize their activities with one another
  - as well as with objects that are purely sequential.
- For example, if two active objects try to send messages to a third object,
  - we must be certain to use some means of mutual exclusion,
  - so that the state of the object being acted on is not corrupted when both active objects try to update its state simultaneously

Concurrency

- This is the point where the ideas of abstraction, encapsulation, and concurrency interact.
- In the presence of concurrency,
  - it is not enough simply to define the methods of an object;
- we must also make certain that the semantics of these methods are preserved
  - in the presence of multiple threads of control

Class Access Specifier

- Private
  - Private members/methods can only be accessed by methods defined as part of the class
  - Data is most often defined as private to prevent direct outside access from other classes
  - Private members can be accessed by members of the class.
- Protected
  - The protected access specifier is needed only when inheritance is involved
  - Protected member/methods are private within a class and are available for private access in the derived class.

Class Access Specifier

- Public
  - The public access specifier allows functions or data to be accessible to other parts of your program
  - Public members/methods can be accessed from anywhere in the program.
  - Class methods are usually public which is used to manipulate the data present in the class.
  - As a general rule, data should not be declared public.
  - Public members can be accessed by members and objects of the class.

Access Specifiers Table

	Within Same Class	In Derived Class	Outside the Class
Private	YES	NO	NO
Protected	YES	YES	NO
Public	YES	YES	YES

Example: Class and Object

```
#include <iostream>
using namespace std;
class student{
   private:
      float cgpa_grade;
      int roll_no;
   public:
      void print_details(){
         cout<< "The details of student are: " <<roll_no
   <<",\t"<< cgpa_grade;
      }
```

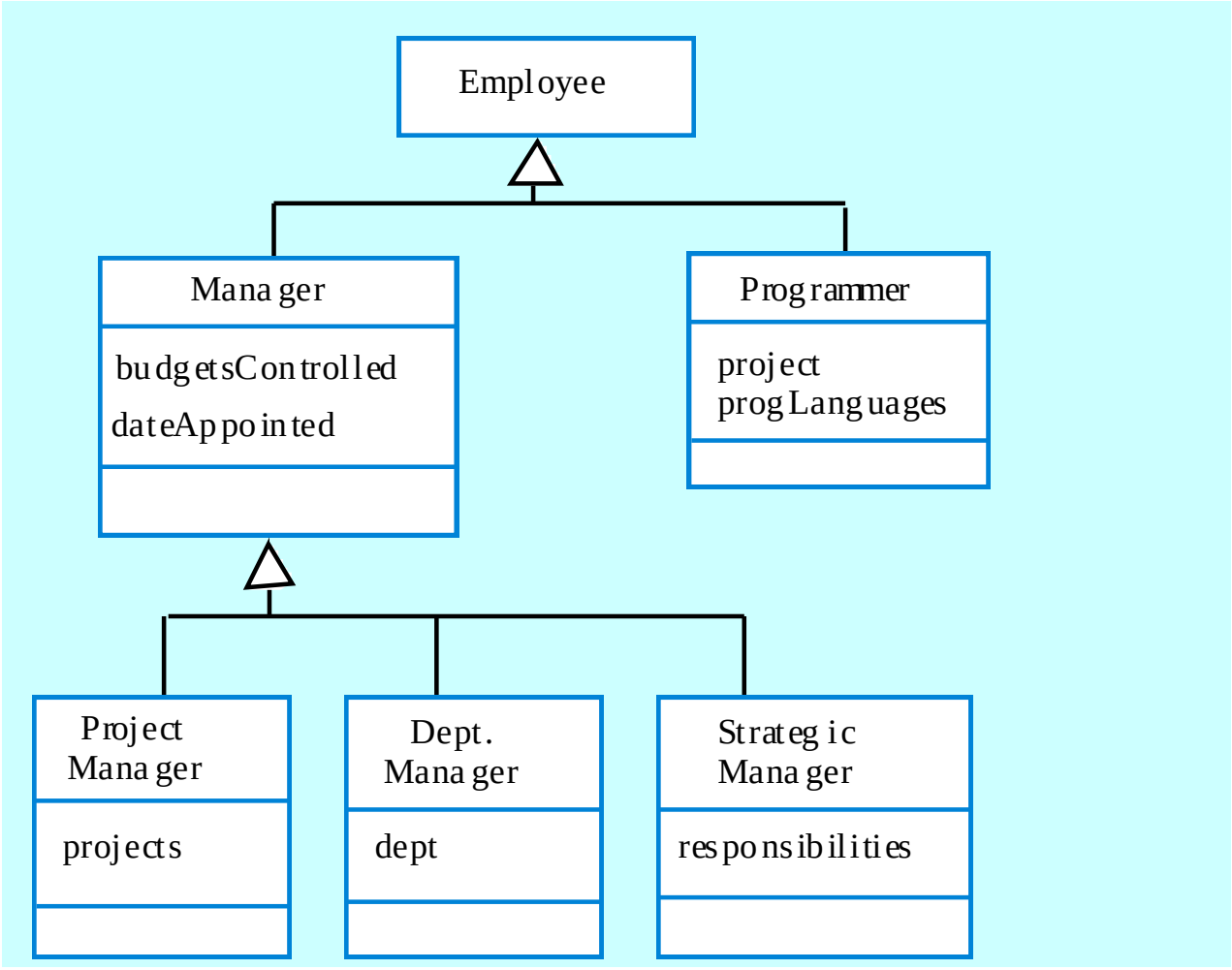
Example: Class and Object

```
void get_details(int r, float f){
        cgpa_grade=f;
        roll_no=r;
      }
};
int main()
{
   student s;
   s.get_details(9878799, 9.7);
   s.print_details();
   cout<<"\n Finished";
   return 0;
}
```

Generalization and Inheritance

- Objects are members of classes that define attribute types and operations.
- Classes may be arranged in a class hierarchy where one class (a super-class) is a generalisation of one or more other classes (sub-classes).
- A sub-class inherits the attributes and operations from its super class and may add new methods or attributes of its own.
- Generalisation in the UML is implemented as inheritance in OO programming languages.

A generalization hierarchy



Advantages of Inheritance

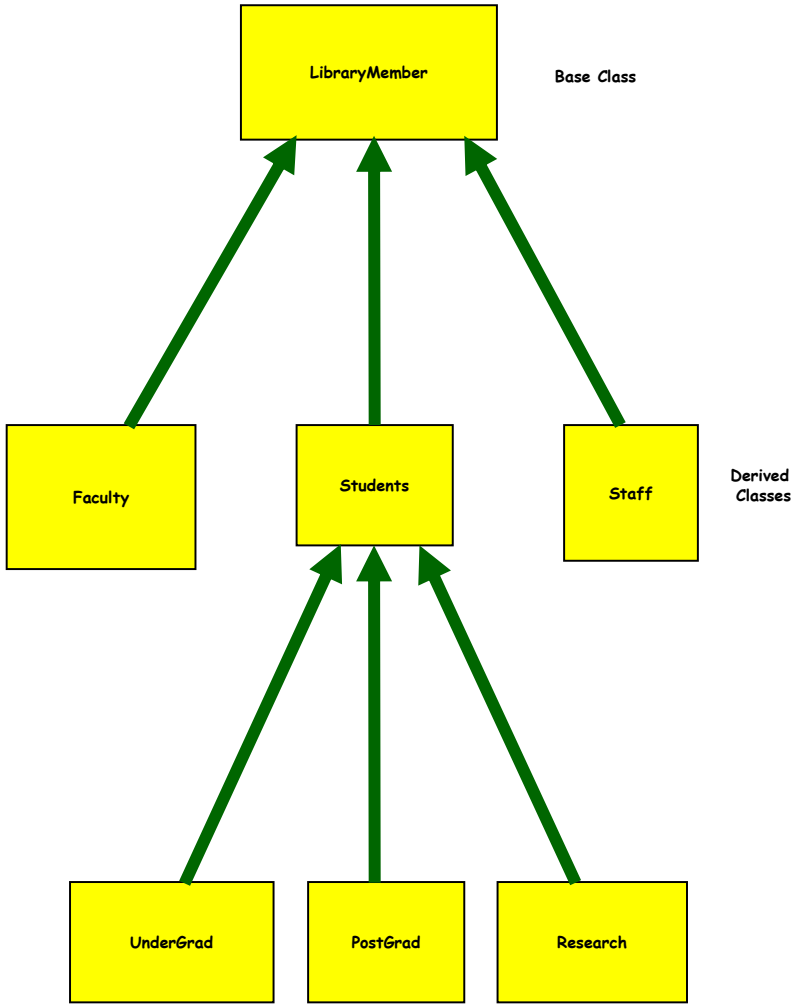
- It is an abstraction mechanism which may be used to classify entities.
- It is a reuse mechanism at both the design and the programming level.
- The inheritance graph is a source of organisational knowledge about domains and systems.

Problems with inheritance

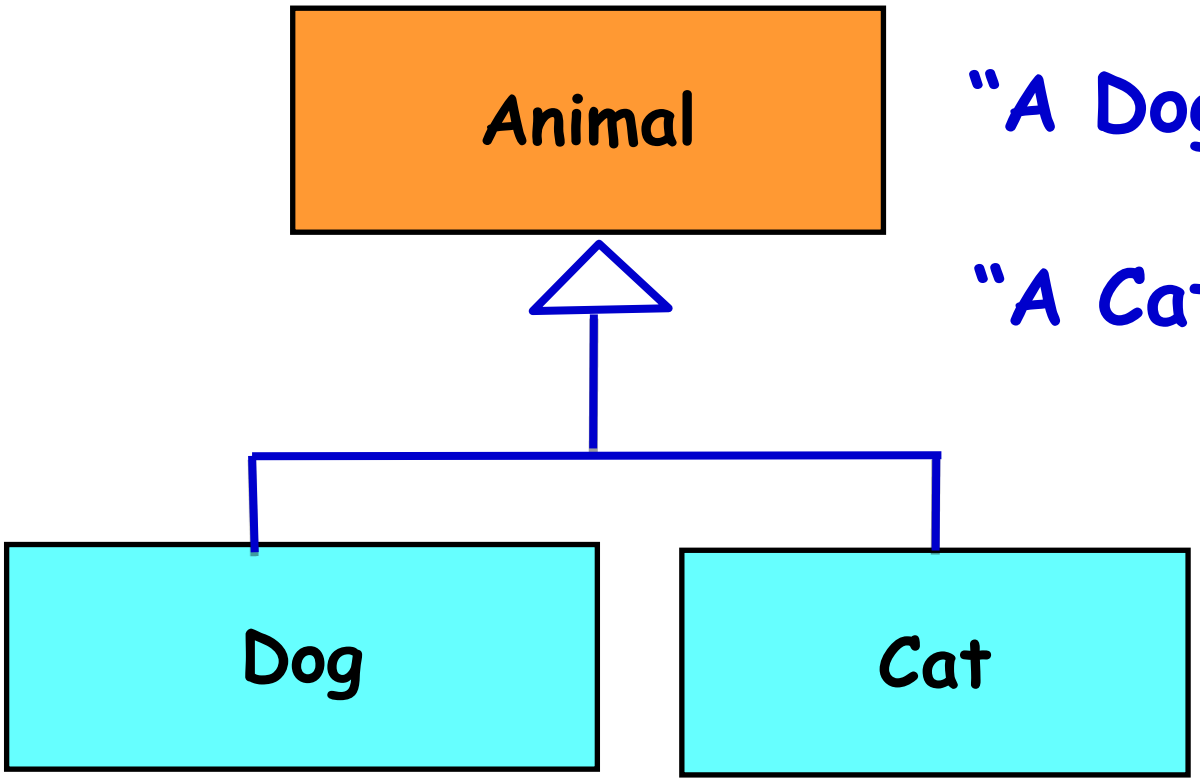
- Object classes are not self-contained and they cannot be understood without reference to their super-classes.
- Designers have a tendency to reuse the inheritance graph created during analysis and it can lead to significant inefficiency.
- The inheritance graphs of analysis, design and implementation have different functions and should be separately maintained.

Inheritance[1]

- Allows to define a new class (derived class) by extending an existing class (base class).
  - Represents generalization-specialization
  - Allows redefinition of the existing methods (method overriding).



Inheritance Example [1]



"A Dog *ISA* Animal"

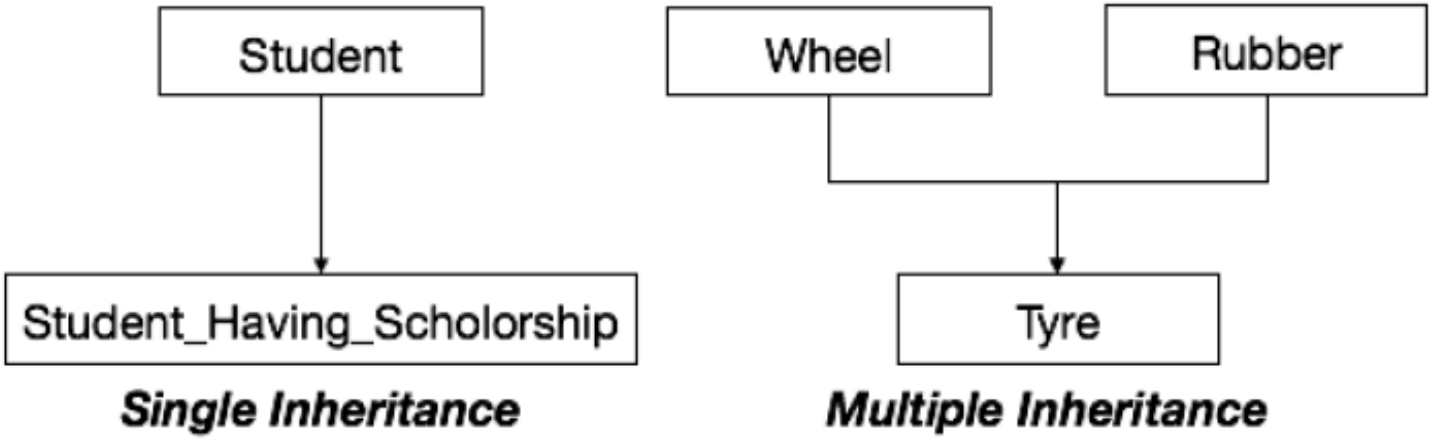
"A Cat *ISA* Animal"

Types of Inheritance

- Single Inheritance
  - A subclass derives from a single super-class.
- Multiple Inheritance
  - A subclass derives from more than one super-classes.
- Multilevel Inheritance
  - A subclass derives from a super-class which in turn is derived from another class and so on.
- Hybrid Inheritance
  - A combination of multiple and multilevel inheritance so as to form a lattice structure.
- Hierarchical Inheritance
  - A class has a number of subclasses each of which may have subsequent subclasses, continuing for a number of levels, so as to form a tree structure.

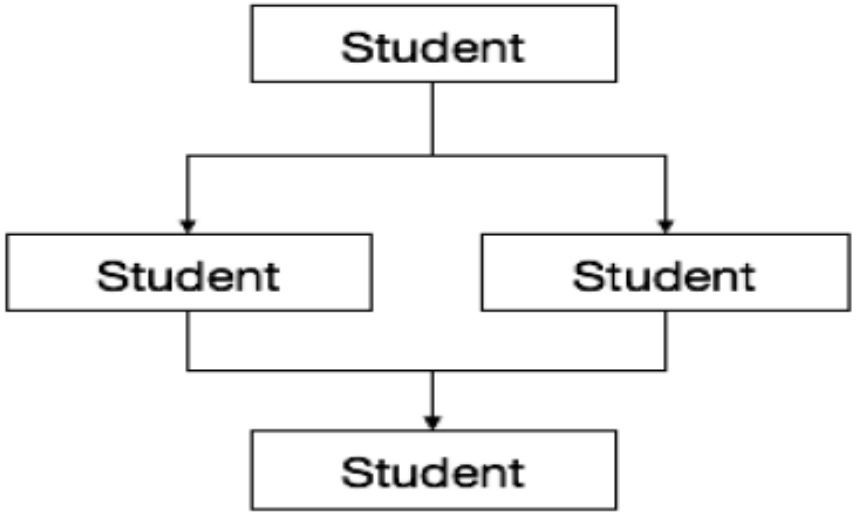
Example of Inheritance

- Single Inheritance
  - See the Example 1 and 2 of Inheritance file
- Multiple Inheritance
  - See the Example 3 of Inheritance file

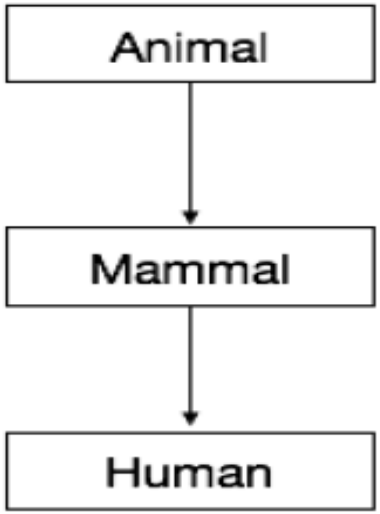


Example of Inheritance

- Hybrid Inheritance
  - See the Example 4 and 5 of Inheritance file
- Multilevel Inheritance
  - See the Example 6 of Inheritance file



Hybrid Inheritance



Multi-level Inheritance

Multilevel vs Multiple Inheritance

- Multiple
  - A class can also be derived from more than one base class, using a **comma-separated list**
  - https://www.w3schools.com/cpp/cpp_inheritance_multiple.asp
- Multilevel
  - A class can also be derived from one class, which is already derived from another class.
  - i.e. MyGrandChild is derived from class MyChild
  - which is derived from MyClass
  - https://www.w3schools.com/cpp/cpp_inheritance_multilevel.asp

Example Multilevel Inheritance

```
// Parent class
class MyClass {
  public:
    void myFunction() {
      cout << "Some content in
     parent class." ;
    }
};
// Child class
class MyChild: public MyClass {
};
```

```
 // Grandchild class
class MyGrandChild: public
     MyChild {
};
int main() {
  MyGrandChild myObj;
  myObj.myFunction();
  return 0;
}
```

Output: Some content in parent class

Example Multiple Inheritance

```
class MyClass {    //Base class
      public:
        void myFunction() {
          cout << "Some content in
    parent class." ;
        }
    };
    // Another base class
    class MyOtherClass {
      public:
        void myOtherFunction() {
          cout << "Some content in
    another class." ;
        }
    };
```

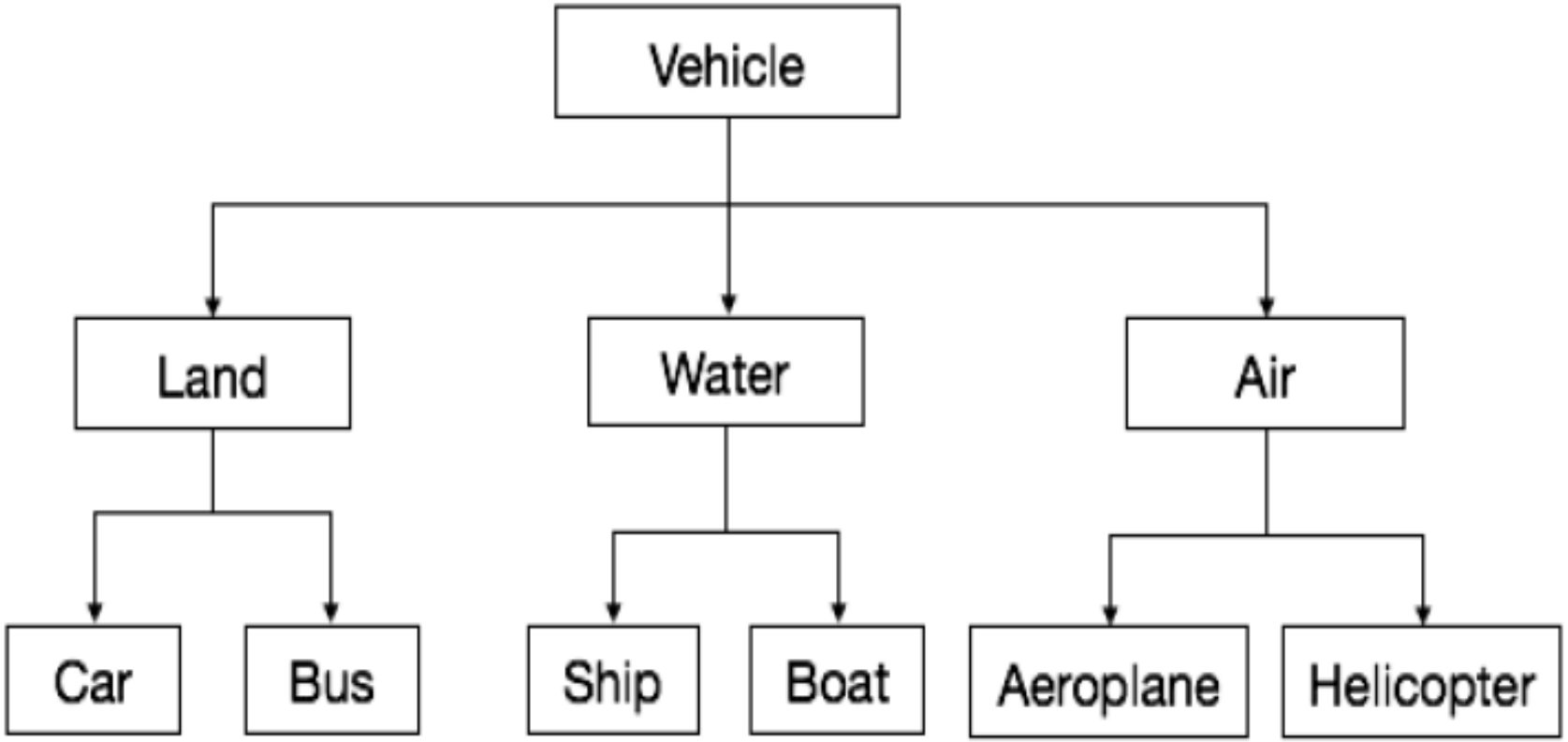
```
// Derived class
    class MyChildClass: public MyCla
    ss, public MyOtherClass {
    };

    int main() {
      MyChildClass myObj;
      myObj.myFunction();
      myObj.myOtherFunction();
      return 0;
    }
```

Output: Some content in parent class

Some content in another class

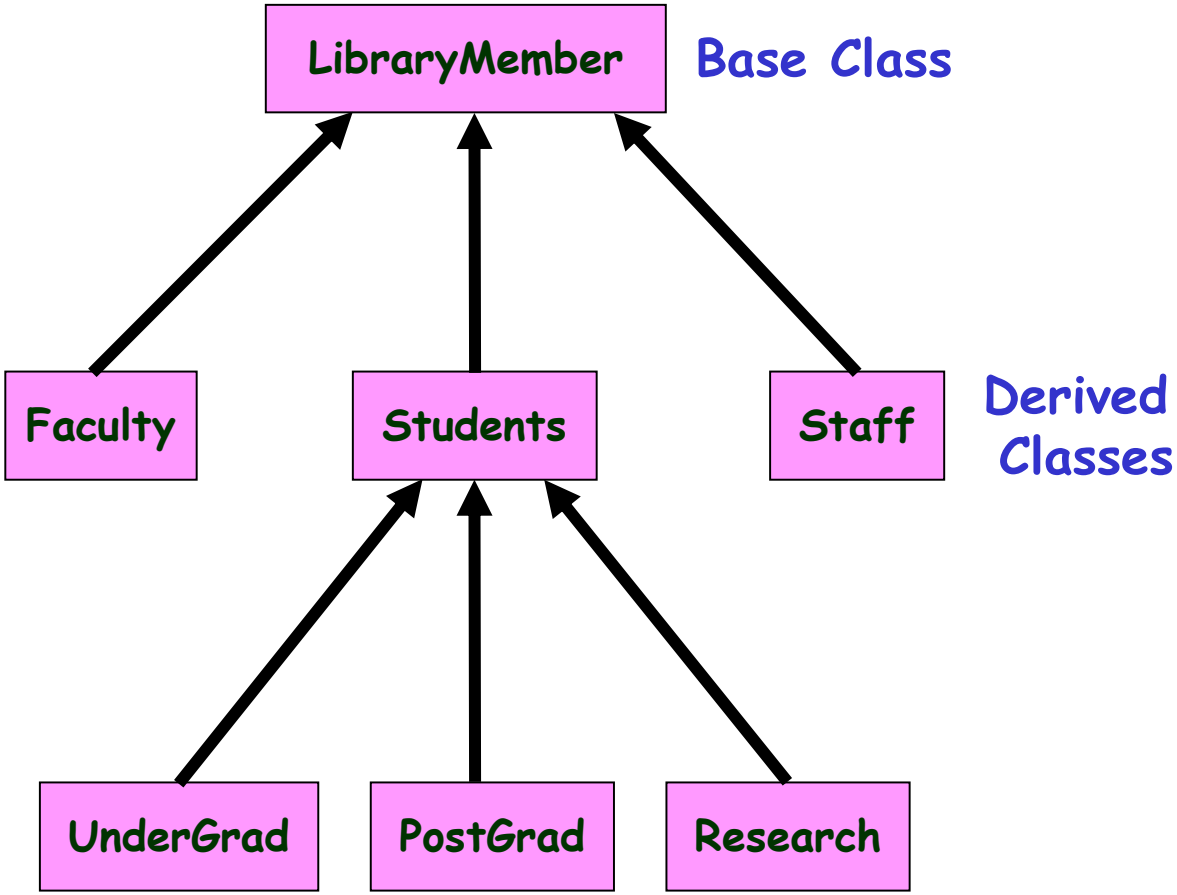
Hierarchical Inheritance



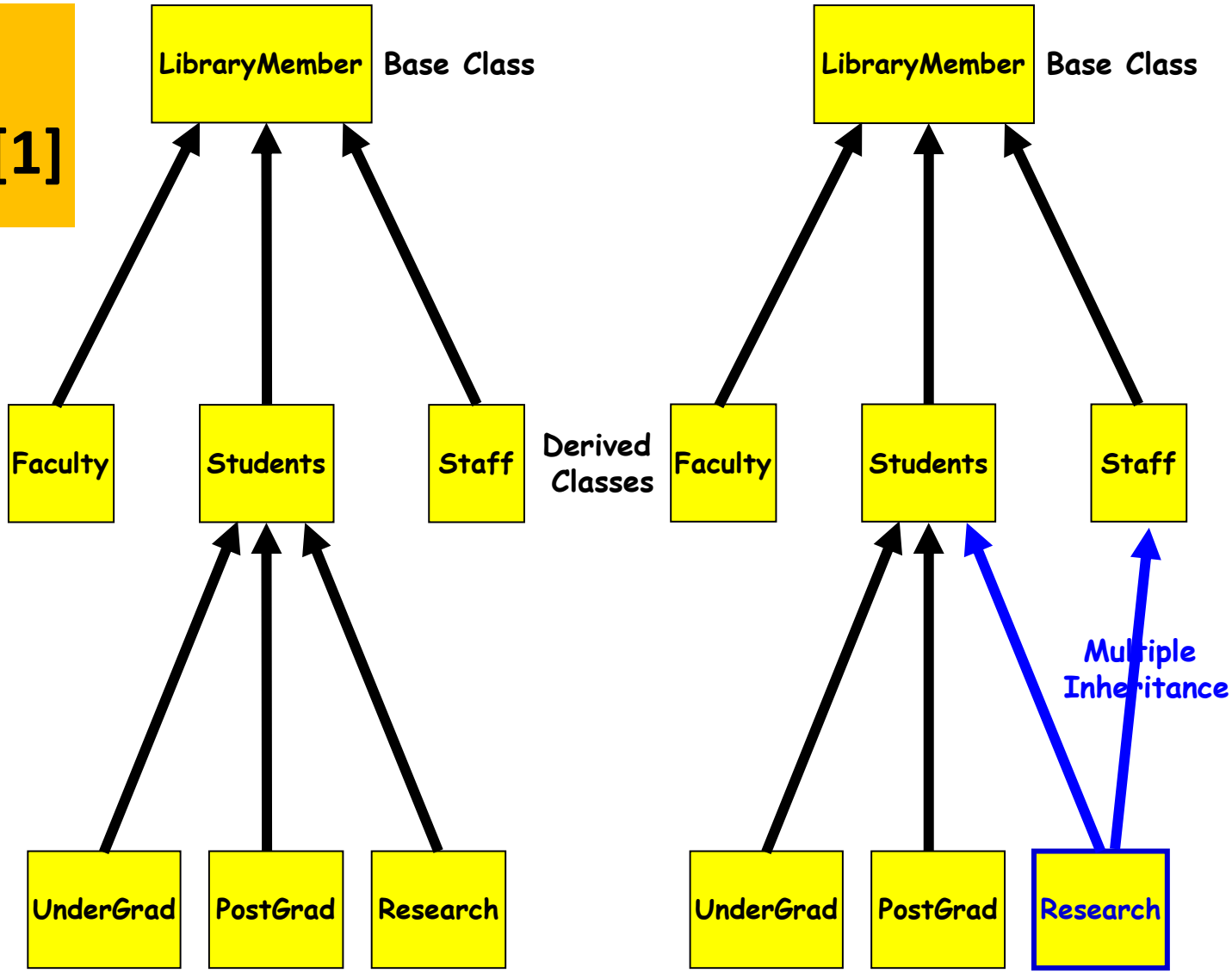
Hierarchical Inheritance

Inheritance [1]

- Lets a subclass inherit attributes and methods from a base class.



Multiple Inheritance[1]

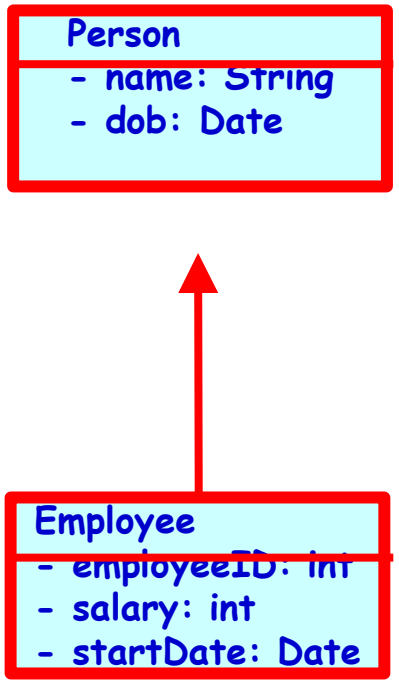


Inheritance Implementation in Java[1]

- Inheritance is declared using the "extends" keyword
  - Even when no inheritance defined, the class implicitly extends a class called Object.

```
class Person{
   private String name;
   private Date dob;
   ...
}
```

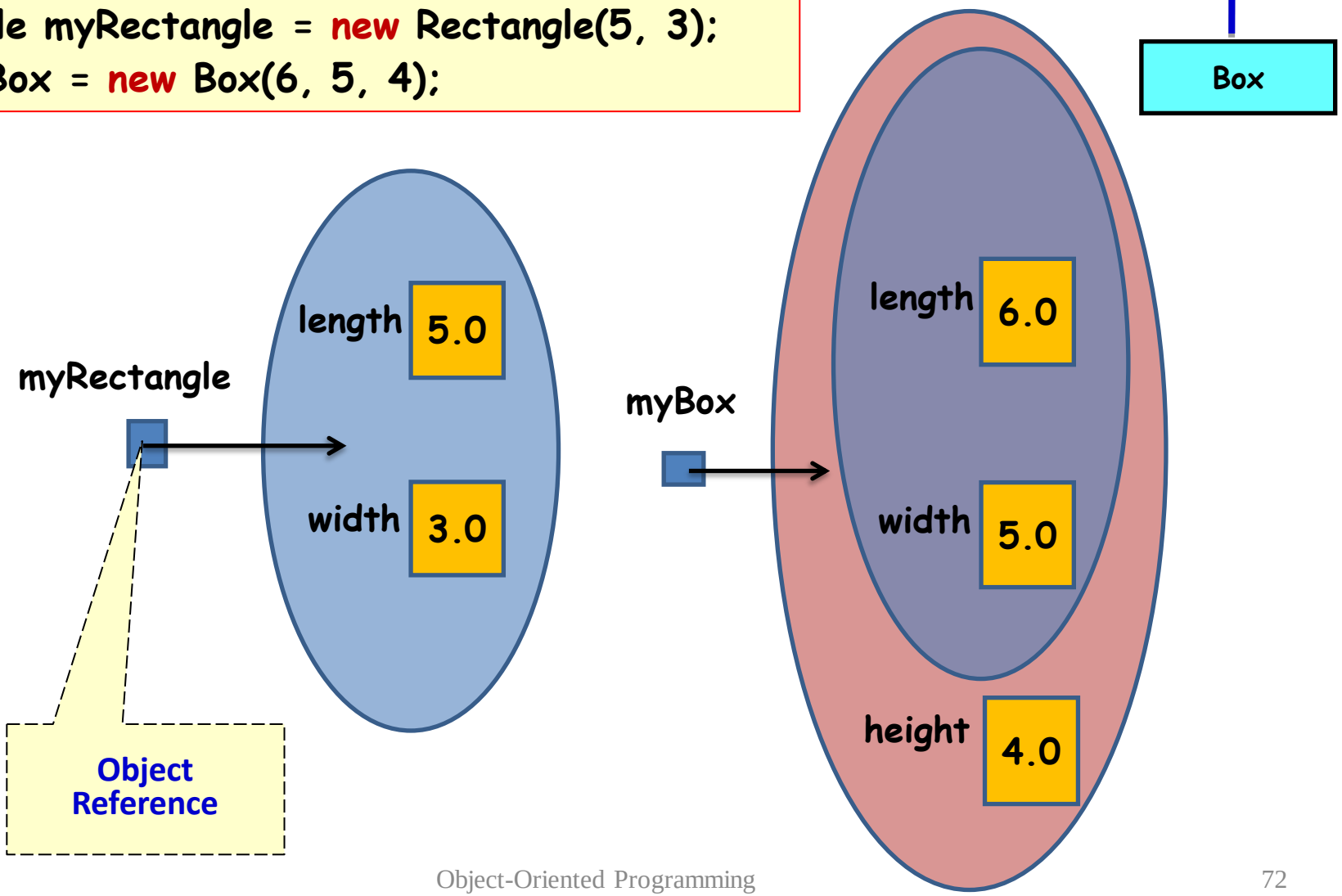
```
class Employee extends Person{
   private int employeeID;
   private int salary;
   private Date startDate;
   ...
}
```



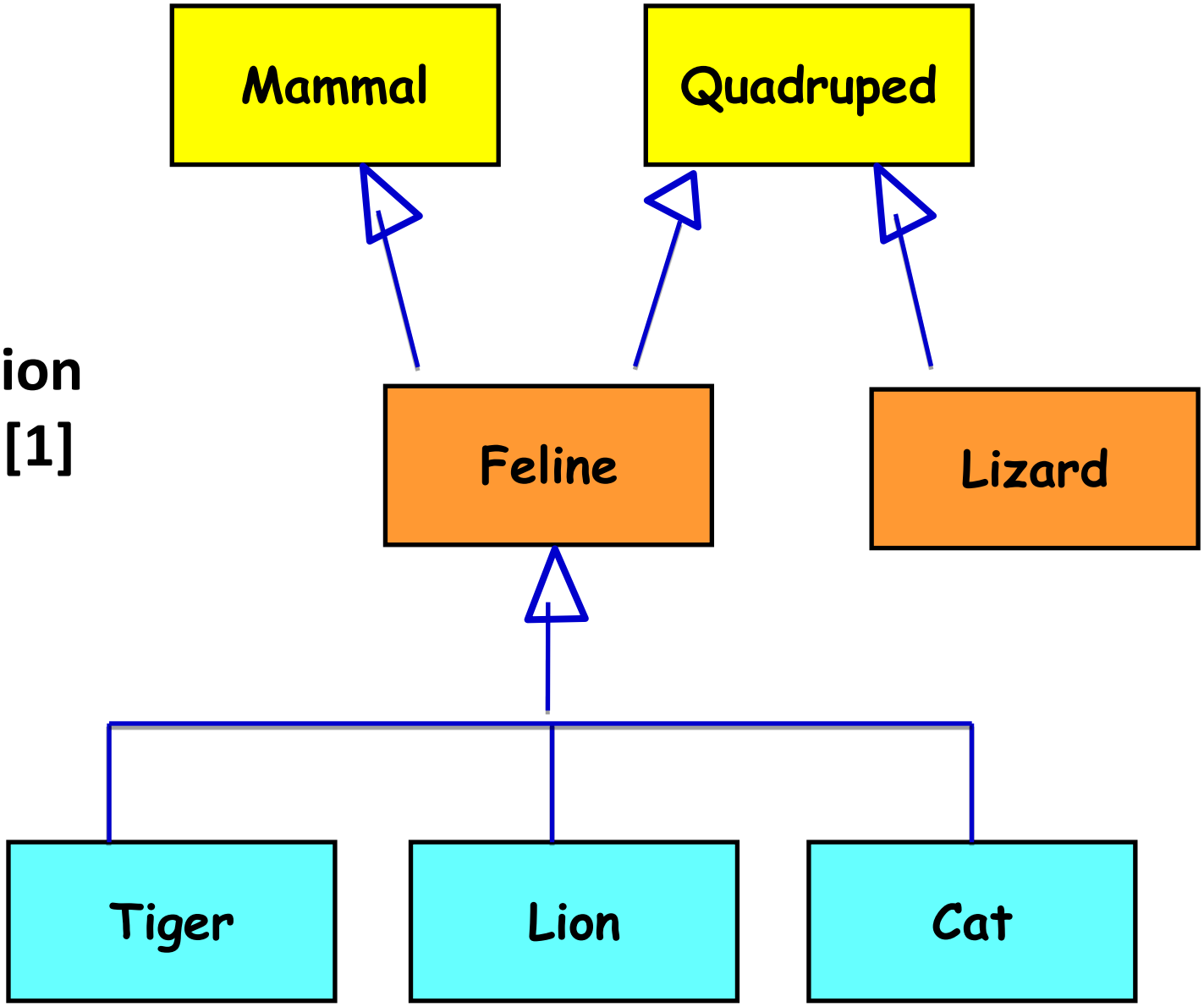

```
Employee anEmployee = new Employee();
```

Objects myRectangle and myBox [1]

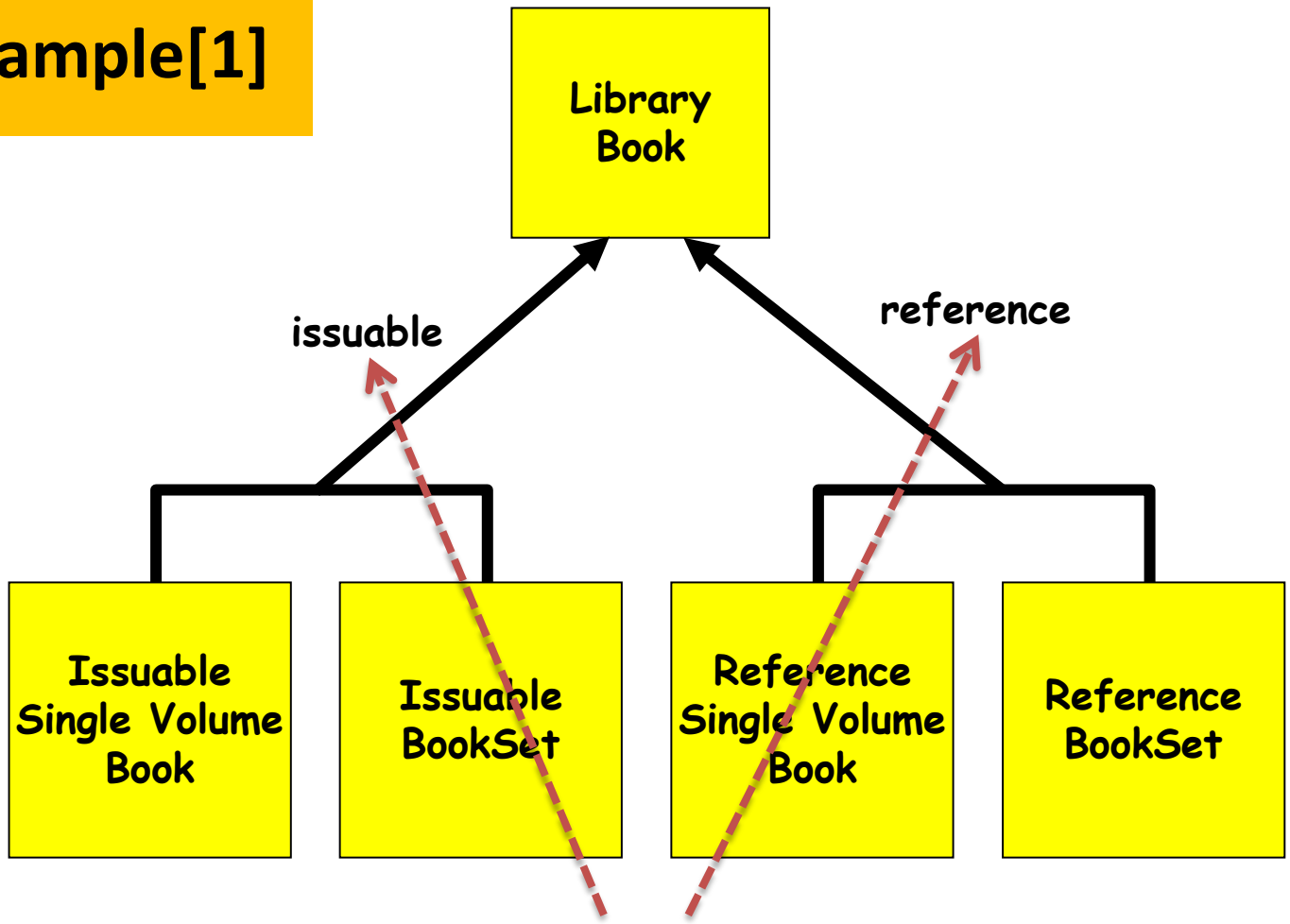
```
Rectangle myRectangle = new Rectangle(5, 3);
Box myBox = new Box(6, 5, 4);
```



More Generalization Examples…[1]



Inheritance Example[1]



Discriminator: allows one to group subclasses into clusters that correspond to a semantic category.

Polymorphism [6]

- **Polymorphism**
  - Polymorphism means "many forms", and it occurs when we have many classes that are related to each other by inheritance.
  - In Inheritance, one class inherit attributes and methods from another class
  - Polymorphism uses those methods to perform different tasks. This allows us to perform a single action in different ways.
- **Compile Time or Early Binding**
  - Function overloading
  - Operator overloading
- **Run Time or Late Binding**
  - Function Overriding
  - Virtual function

Early Binding [6]

- Early binding refers to events that occur at compile time.
- It occurs when all information needed to call a function is known at compile time.
  - It means that an object and a function call are bound during compilation.
  - i.e. normal function calls, standard library functions, overloaded function calls, and overloaded operators.
- In this, all information necessary to call a function is determined at compile time that leads to
  - Increase the efficiency and
  - function calls are very fast.

Late Binding [6]

- Late binding refers to function calls that are not resolved until run time.
  - virtual functions are used to achieve late binding.
  - actually called is determined by the type of object pointed to by the pointer when access is via a base pointer or reference
- In most cases this cannot be determined at compile time
  - the object and the function are not linked until run time.
- The main advantage to late binding is flexibility.
  - a function call is not resolved until run time
- The disadvantage of late binding/dynamic binding
  - late binding leads to slower execution times.

Function Overloading

- Function overloading is the process of using the same name for two or more functions
- The secret to overloading is that each redefinition of the function must use either different types of parameters or a different number of parameters.
- It is only through these differences that the compiler knows which function to call in any given situation
- Therefore, whenever a binding of static, private and final methods (in Java) happen, type of the class is determined by the compiler at compile time and the binding happens then and there.
  - The reason is that these method cannot be overridden, and the type of the class is determined at the compile time

Example of Function Overloading

```
#include <iostream>
using namespace std;
int myfunc (int i);     // these differ in types
                                //of parameters
double myfunc (double i);
int main ()
{
 cout << myfunc (10) << " ";
                               // calls myfunc(int i)
  cout << myfunc (5.4);
                             // calls myfunc(double i)
  return 0;
}
```

```
double
myfunc (double i)
{
  return i;
}
int
myfunc (int i)
{
return i;
}
```

Function Overriding

- Method Overriding is a perfect example of dynamic binding
  - In overriding both parent and child classes have same method and in this case the type of the object determines which method is to be executed
- The method definition and method call are linked during the run time.
  - Program execution is slower
- For example, think of a base class called Animal that has a method called animalSound().
  - Derived classes of Animals could be Pigs, Cats, Dog..
- Now we can create Pig and Dog objects and override the animalSound() method.
  - https://www.w3schools.com/cpp/cpp_polymorphism.asp

Example of Function Overriding

```
class Animal {  // Base class
    public:
      void animalSound() {
      cout << "The animal makes a
  sound \n" ;
    }
  };    // Derived class
  class Pig : public Animal {
    public:
      void animalSound() {
      cout << "The pig says: wee
  wee \n" ;
     }
  };
```

```
// Derived class
    class Dog : public Animal {
      public:
        void animalSound() {cout << "The
    dog says: bow wow \n" ;   }
    };
    int main() {
      Animal myAnimal;
      Pig myPig;
     Dog myDog;
      myAnimal.animalSound();
      myPig.animalSound();
      myDog.animalSound();
      return 0;
    }
```

Function Overloading VS Function Overriding

- In the context of Inheritance, overriding of functions occurs when one class is inherited from another class.
  - Overloading can occur without inheritance.
- **Function Signature**
  - Overloaded functions must differ in function signature i.e. either number of parameters or type of parameters should differ.
  - In overriding, function signatures must be same in the context of virtual function
- **Scope of functions**
  - Overridden functions are in different scopes; whereas overloaded functions are in same scope.
- **Behavior of functions:**
  - Overriding is needed when derived class function has to do some added or different job than the base class function.
  - Overloading is used to have same name functions which behave differently depending upon parameters passed to them.
- Reference- https://www.geeksforgeeks.org/function-overloading-vs-function-overriding-in-cpp/

Constructor and Destructor [6]

- Constructor
  - A constructor in C++ is a **special method** that is automatically called when an object of a class is created.
  - To create a constructor, use the same name as the class, followed by parentheses ()
- Destructor
  - Like constructor, destructor is also a member function of a class that has the same name as the class name preceded by a tilde(~) operator.
  - It helps to deallocate the memory of an object.
  - It is called while object of the class is freed or deleted

Example of Constructor

```
#include<iostream>
using namespace std;
class MyClass {     // The class
      public:          // Access specifier
        MyClass() {     // Constructor
          cout << "Hello World!";
        }
    };

    int main() {
      MyClass myObj;
 // Create an object of MyClass (this will call the constructor)
      return 0;  }
```

Example of Destructor

```
#include<iostream>
using namespace std;
class Z
{ public:
   Z()    // constructor
     {     cout<<"Constructor called"<<endl;    }
  ~Z()   // destructor
    {     cout<<"Destructor called"<<endl;    } };
int main()
{    Z z1;   // Constructor Called
   int a = 1;
   if(a==1)    {       Z z2;  // Constructor Called
                    }  // Destructor Called for z2
} //  Destructor called for z1
```

Source: https://www.geeksforgeeks.org/difference-between-constructor-and-destructor-in-c/

Copy Constructor [6]

- Copy Constructor
  - More important part of overloaded constructor
  - A copy constructor is a member function which initializes an object using another object of the same class.
  - A copy constructor has the following general function prototype
    - ClassName (const ClassName &old_obj);
- A copy constructor is called
  - When an object of the class is returned by value and passed (to a function) by value as an argument.
  - When an object is constructed based on another object of the same class.
  - When the compiler generates a temporary object.

Operator Overloading [6]

- An operator function defines the operations that the overloaded operator will perform relative to the class upon which it will work.
- An operator function is created using the keyword operator.
- operator+( ) has only one parameter even though it overloads the binary + operator.
- You might expect two parameters corresponding to the two operands of a binary operator

Operator Overloading [6]

- Operator functions return an object of the class they operate on while ret-type can be any valid type.
- When you create an operator function, substitute the operator for the # that is a placeholder
- A member operator function takes this general form:

```
ret-type class-name::operator#(arg-list)
{     // operations }
```

Virtual Function [6]

- A virtual function is a member function that is declared within a base class and redefined by a derived class
- The derived class redefines the virtual function to fit its own needs
- Virtual functions implement the "one interface, multiple methods" that underlies polymorphism.
- The virtual function within the base class defines the form of the interface to that function.
- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class.

Virtual Function [6]

- Virtual functions behave just like any other type of class member function when it is accessed.
- When a base pointer points to a derived object that contains a virtual function, it determines which version of that function to call based upon the type of object pointed to by the pointer.
- This determination is made at run time.
- Thus, when different objects are pointed to, different versions of the virtual function are executed.
- The same effect applies to base-class references.

Pure Virtual Function [6]

- A pure virtual function is a virtual function that has no definition within the base class.
- To declare a pure virtual function, use this general form:
  - virtual type func-name(parameter-list) = 0;
  - i.e. virtual void show() = 0;
- A class that contains at least one pure virtual function is said to be abstract.
- No objects of an abstract class may be created as an abstract class contains one or more functions with no definition.

Exception Handling

- Exceptions
  - It provides a systematic object-oriented approach to handle run-time errors generated by classes
  - Exceptions are errors that occur at run time
  - It is called by variety of exceptional circumstance such as
    - Running out of memory
    - Not being able to open a file
    - Trying to initialize an object to an impossible value
    - Using an out-of-bounds index to a vector

Exception Handling

- Exception handling
  - It facilitates solution to programmer such that our program can automatically invoke an error-handling routine when an error occurs.
  - It allows to manage run-time errors in an orderly fashion.
  - Exception handling is built upon three keywords:
    - Try :- code is monitored for exception or error
    - Catch :- exception is caught using catch and processed
    - Throw :- if an exception occurs within the try block then it is thrown using throw.

general form of try and catch

```
try {
    // try block }
catch (type1 arg) {
     // catch block }
catch (type2 arg) { // catch block }
catch (type3 arg) { // catch block }
. . .
catch (typeN arg) { // catch block }
```

throw statement

- The general form of the throw statement is shown here:
  - throw *exception*;
- throw generates the exception specified by exception.
- If this exception is to be caught, then throw must be executed either from within a try block itself, or from any function called from within the try block

A simple exception handling example

```
#include<iostream>
using namespace std;
int main()
{ cout << "Start\n";
try {
// start a try block
cout << "Inside try block\n";
throw 100; // throw an error
cout << "This will not execute";
}
catch (int i)
{ // catch an error
cout << "Caught an exception --
value is: ";
cout << i << "\n"; }
cout << “end";
return 0; }
```

Catching All Exceptions

```
#include<iostream>
using namespace std;
void Xhandler(int test)
{ try
{ if(test==0) throw test; // throw int
if(test==1) throw 'a'; // throw char
if(test==2) throw 123.23;
 // throw double }
catch(...) {
// catch all exceptions
cout << "Caught One!\n"; }
}
```

```
int main() {
cout << "Start\n";
Xhandler(0);
Xhandler(1);
Xhandler(2);
cout << "End";
return 0;
}
```

Object Containers

- The container is **an object that contains other objects**.
- The container can contain all other sheet objects.
- The objects are grouped together and have common settings for font, layout and caption.
- It can also be accessed from the Object menu, when the container is the active object.

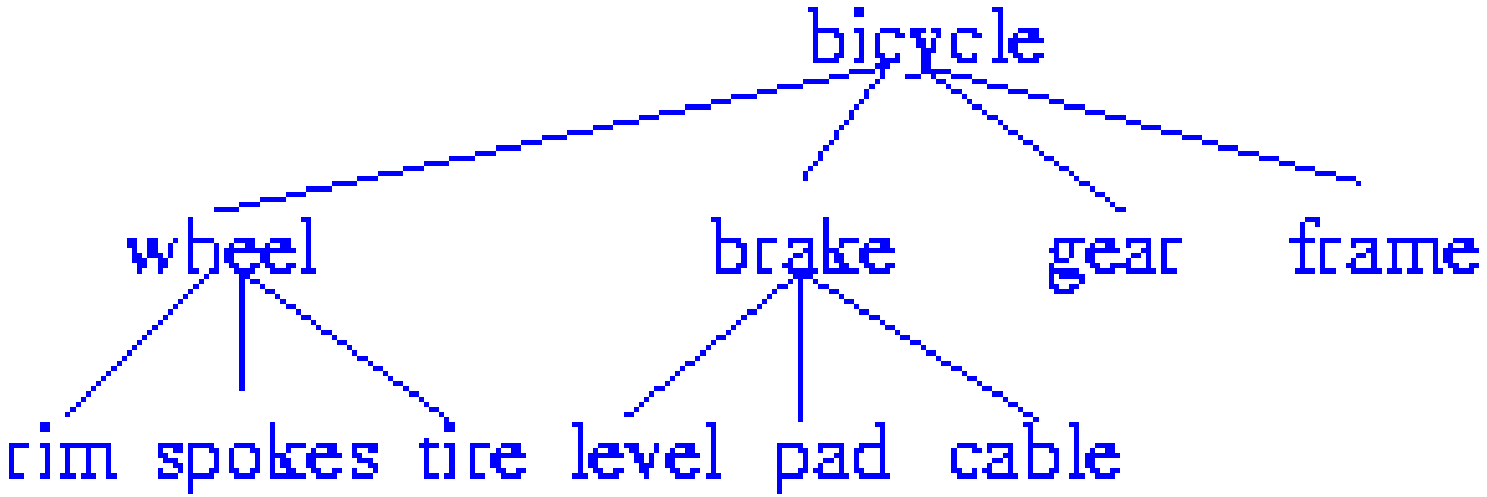
Storing Objects and Primitives

- ***A container can store instances of any Java class*.**
  - For example, you can store a String, a Date, and an Integer in the same container.
  - When you retrieve an object, you must cast the object back to the required type before applying a type-specific method. Java's dynamic cast-checking mechanism ensures that any attempt to cast an object to the wrong type causes a runtime exception.
- ***A container cannot directly store a primitive***.
  - Primitives can only be stored in containers using a special technique described by Storing Primitives.

Object Containment

- Objects that contain other objects are called *complex* or *composite* objects.
- Containment indicates that one object contains another.
- There can be multiple levels of containment, forming a *containment hierarchy* among objects.
- Example: A bicycle design database

Object Containment



Object Containment

- Containment allows data to be viewed at different granularities by different users.
  - E.g., *wheel* by wheel designer but *bicycle* by a sales-person. The containment hierarchy is used to find all objects contained in a *bicycle* object.
- In certain applications, an object may be contained in several objects.
- In such cases, the containment relationship is represented by a DAG rather than by a hierarchy.
- Video Lectures:
  - https://www.youtube.com/watch?v=TxcSDAbrdpI
  - https://www.youtube.com/watch?v=87Ty18RNr-c

Object Containment

- Hierarchy of contained classes means one object nested within the other object or class.
- The CreditCard object contains a BillingInfo object, which in turn contains an Address object.
- Address is an object that could be used by several other classes, such as Order ShippingInfo, or Invoice.
- BillingInfo is restricted to the payment and should not exist outside of it.
- The exception to this rule is the various collection classes that contain a type of an object.
- In this case, the container's primary behavior is to insert, remove, and maintain the objects it contains.

Object Identity

- An object retains its identity even if some or all the values of variables or definitions of methods change over time.
  - With object identity, objects can contain or refer to other objects.
  - Identity is a property of an object that distinguishes the object from all other objects in the application
- This concept of object identity does not apply to tuples of a relational database.
- Several forms of identity:
  - **value**: A data value is used for identity (e.g., the primary key of a tuple in a relational database).
  - **name**: A user-supplied name is used for identity (e.g., file name in a file system).
  - **built-in**: A notion of identity is built-into the data model or programming languages, and no user-supplied identifier is required (e.g., in OO systems).

Object Identity

- Object identity is typically implemented via a *unique*, *system-generated* OID.
  - The value of the OID is not visible to the external user but is used internally by the system to identify each object uniquely and to create and manage inter-object references.
  - There are many situations where having the system generate identifiers automatically is a benefit, since it frees humans from performing that task.
  - System-generated identifiers are usually specific to the system and must be translated if data are moved to a different database system.
  - System-generated identifiers may be redundant if the entities being modeled already have unique identifiers external to the system, e.g., SIN#.

The Meaning of Persistence

- Introducing the concept of persistence to the object model
  - It gives rise to object-oriented databases
- Persistence saves the state and class of an object across time or space
- Persistence may be achieved through a modest number of commercially available object-oriented databases
- A more typical approach to persistence is to provide an object-oriented skin over a relational database.
  - Customized object-relational mappings can be created by the individual developer

The Meaning of Persistence

- Persistence deals with more than just the lifetime of data.
- In object-oriented databases, not only does the state of an object persist, but its class must also transcend any individual program
  - so that every program interprets this saved state in the same way.
- Persistence is the property of an object through which its existence
  - transcends time i.e., the object continues to exist after its creator ceases to exist and/or
  - space (i.e., the object's location moves from the address space in which it was created).

Persistence

- An object occupies a memory space and exists for a particular period.
- In traditional programming, the lifespan of an object was typically the lifespan of the execution of the program that created it.
- In files or databases, the object lifespan is longer than the duration of the process creating the object.
- This property by which an object continues to exist even after its creator ceases to exist is known as persistence.

Persistence

- The data manipulated by an object-oriented database can be either transient or persistent.
- Transient data is only valid inside a program or transaction; it is lost once the program or transaction terminates.
- Persistent data is stored outside of a transaction context, and so survives transaction updates.
- Usually, the term persistent data is used to indicate the databases that are shared, accessed and updated across transactions.
- The two main strategies used to create and identify persistent objects are:
  - Persistence extensions
  - Persistence through reachability

References

1. Rajib Mall, “Fundamentals of Software Engineering”, 3rd edition, PHI, 2009.
2. R.S. Pressman, “Software Engineering: A Practitioner's Approach”, 7th Edition, McGraw-Hill.
3. Sommerville, “ Introduction to Software Engineering”, 8th Edition, Addison-Wesley, 2007.
4. Grady Booch et al, Object-Oriented Analysis and Design with Applications, Addison Wesley, 2007.
5. https://www.edureka.co/blog/data-hiding-in-cpp/
6. Herbert Schildt, “The Complete Reference C++”, Fourth Edition, Tata McGraw-Hill, New Delhi, 2003.

THANK YOU