



Object Oriented Programming

1 Introduction

2 Objects & Class

3 Abstraction & Encapsulation

4 Inheritance

5 Polymorphism



Introduction

Module I

Objectives



At the end of this module, you will be able to:

- Describe complexity involved in developing Software Application
- Understand need for modularity in a Software Application
- Describe need for Divide and Conquer approach
- Compare Procedure oriented with Object Oriented approach
- Define object oriented programming

Duration: 2 hrs

- Complexity is directly proportional to Scope of Application. Making it challenging.
- Failure to address Complexity will lead to *Software Crisis*. **Consequences being, over budget, non capture of requirements, late delivery, inefficient execution and so on...**
- Addressing Software Complexity
 - Program being divided into small functions
 - Structured approaches like Top-Down and Bottom-Up approaches being used to simplify development.
 - Analyzing Customer Requirement as set of specific tasks were encapsulated into simple logical blocks called Procedures.

- The act of partitioning a system into individual components
- Helps reduce complexity to some extent.
- Creates a number of well-defined documented boundaries within the system.
- Thus developer need to balance two competing technical concerns
 - *the desire to encapsulate abstractions*
 - *the need to make certain abstractions visible to other modules*

- A subprogram is also known as a procedure or a function
- Algorithms of functions are designed keeping data representation in mind
- Any change in strategy relating to data representation will directly impact functions that are dependent on data
- These dependent functions would have no option but to reconfigure their logic to suit modified data representation.
- Contributors
 - Dijkstra – Layers of abstraction
 - Niklaus Wirth – Program Development by Stepwise Refinement
 - Parnas – Information hiding

- While arriving at solution for a requirement, it is essential to decompose problem into smaller manageable chunks of code
- Smaller chunks are modules
 - Termed in Software terminology as “Divide and Rule”
- Benefits
 - Complexity of a Application can be simplified
 - Refinement can be achieved independently specific to modular level

Divide and Rule (Contd.).



Two prominent paradigms

- Algorithmic Decomposition
 - Top down structure followed
 - Focus on Data Flow
 - Emphasis on Procedure – “What is happening” approach

- Object Oriented Decomposition
 - Decomposition of System on key abstractions in problem domain.
 - Each Object/ abstraction maps to a real world entity and has a unique behavior.
 - Objects interact with each other through messages
 - Emphasis on Object – “Who is getting affected” approach

Object-Oriented Programming - Point of View



- OOP is a paradigm with certain goals:
 - Intuitive program decomposition.
 - Modularity.
 - Code reuse (related to flexibility, maintainability...)
- Look at how it addresses those goals:
 - ...in programming language design.
 - ...in software development methodology
- And look at how you as a programmer would contribute:
 - ...through good programming practices.
 - ...by knowing and following design patterns.

Object-Oriented Programming



- Object -Oriented Programming is a programming paradigm that organizes around objects and data with following salient features:
 - Groups together data (attributes) and functions (methods) to create re-useable objects
 - Serves as an approach towards programming where problem domain is viewed in a natural way
 - Views a problem as a collection of interacting objects with certain features/attributes and certain behaviors.

- It helps a great deal if the programme being developed is analogous to a real world solution to the problem.
 - requirements are easier to understand,
 - debugging eased by access to real world solution
 - system is less likely to be obsolete in the end.

First adopted by simulation programmers - 1960.

Conventional Vs Object-Oriented Approach

Procedure (Conventional) Approach

- Emphasis on algorithms
 - Functions
 - Global data
- Top down approach

	Design approach	
	Conventional	Object-oriented
Goal	identify major functions	identify major objects
Result	gather radar info update display	planes display screen radar receiver

Object-oriented Approach

- Emphasis on data abstraction
 - Objects
 - Functions and data are grouped
- Bottom up approach

Object plane:
data (identification, location, altitude, direction and speed)
+
functionalities (change of location, altitude, direction, speed, displayed on the air traffic control screen)

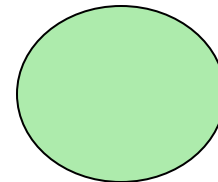
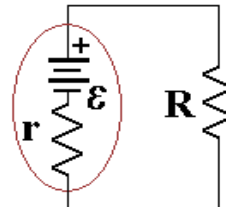
Object-Oriented System View



- Objects collaborate to provide higher-level system functionality
- System decomposition is done on basis of objects, and not data & functions as in structured approach
- In structured methodology, developers build complex systems using algorithms or functions as basic building block
- In object-oriented methodology, developers build complex systems using objects as basic building block

Object-Oriented World View

- Program consists at run time, a collection of interconnected, interacting components (“**objects**”).
- Objects interact by sending each other “**messages.**” (And no other way.)
- Different objects may respond to same message in completely different ways. (Even objects of the same “type”.) **Or maybe not**
 - Tangible Things as a car, printer, ...
 - Roles as employee, boss, ...
 - Incidents as flight, overflow, ...
 - Interactions as contract, sale, ...
 - Specifications as colour, shape, ...



In this module, we discussed:

- Software Application complexity
- Modularity in a Software Application
- Divide and Conquer approach
- Procedure oriented Vs Object Oriented approach
- Object-oriented programming and objects

Objects & Class

Module 2

Objectives



At the end of this module, you will be able to:

- Define objects and class
- Identify State and Behavior of Objects
- Identify principles of Object Oriented Programming

Duration: 2 hrs

What is an Object?



- Real World objects are things that have
 - State
 - Behavior
 - Ex. A Dog
 - State – Name, Color, Breed
 - Behavior – Barking, Waging Tail
- An *Object* in Software is a bundle of variables (state) and related methods (operations)
- Simulation programmers have always viewed the world as objects and semantic relationships, and this led to a natural programming model consisting of objects with
 - State (internal data).
 - Behaviour (attributes or methods) and
 - Identity (object names or references)

An Example of an Object



Play

State / Attributes

Length
Breadth
Height
Weight
Colour

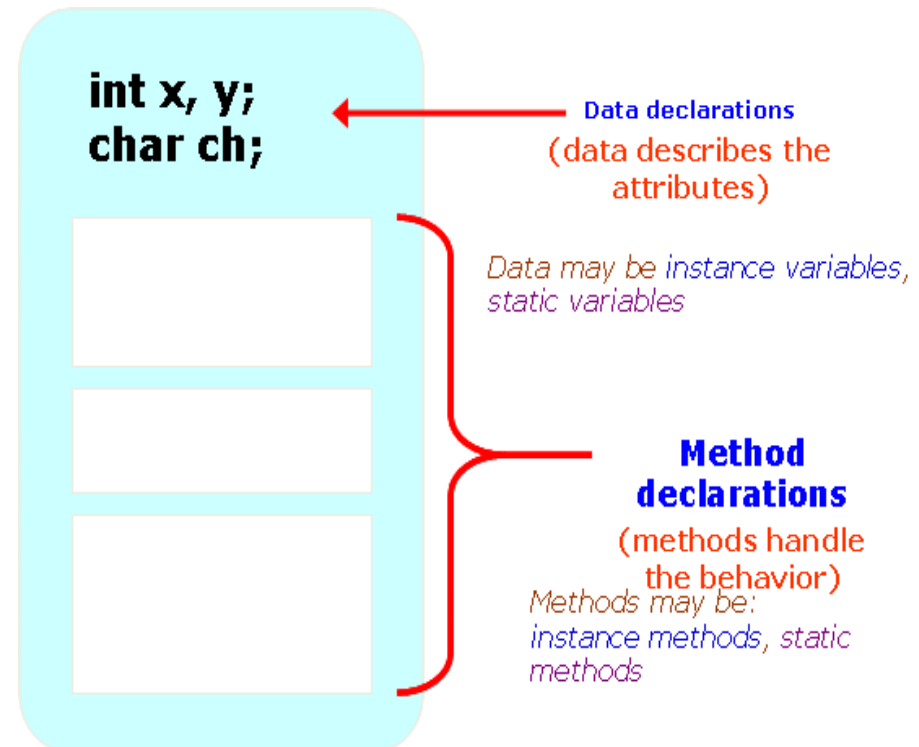
Behaviours / method

Play
Rewind
Forward Fast
Record
Stop/Eject

Classes

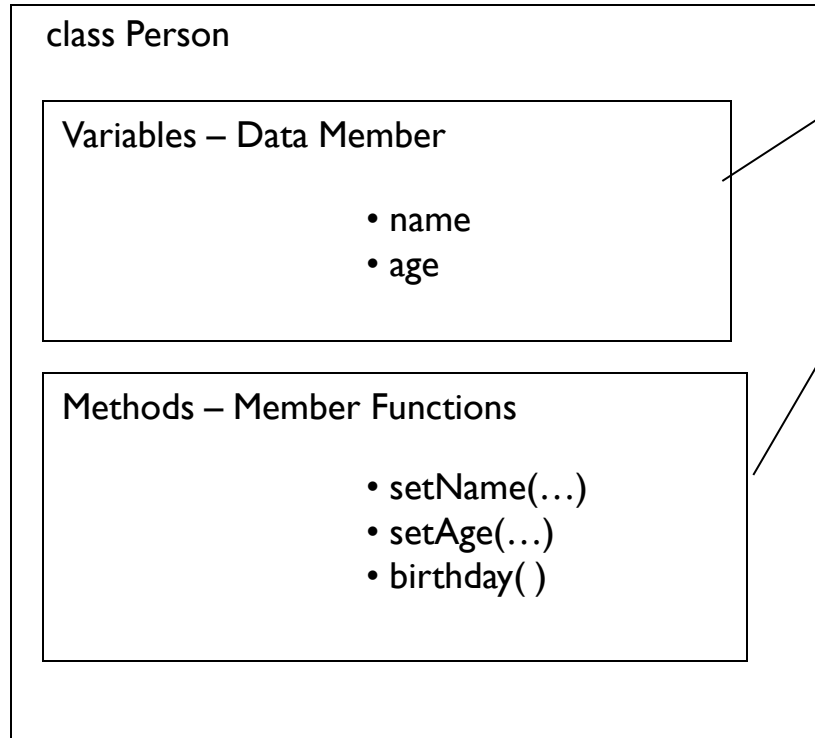
- Classes are Template (“Logical Template”), Blue Print used to create Object
- A class is a general description of a set of objects with common attributes and common behaviors'
- Classes generally tend to be related in the form of a hierarchy. Classes at a certain level in the hierarchy are derived from the classes at the immediately preceding level
- However class at top level hierarchy shall not have a super class. Such classes are called Generic classes in common

A class contains **data declarations** and **method declarations**



Class – A code snippet

Define a class Person with **data (instance variables)** and **methods (instance methods)**



```
class Person {
```

```
    char name;  
    int age;
```

```
    void setName (char n) {  
        name = n;  
    }  
    void setAge (int a) {  
        age = a;  
    }  
    void birthday ( ) {  
        age++;  
        cout<< name<<"is now'"<<age;  
    }  
}
```

Object Creation

- The *new* keyword is used to explicitly create object that is referenced

```
Person p = new Person();
```

Object reference

object

- Constructors are invoked using *new*

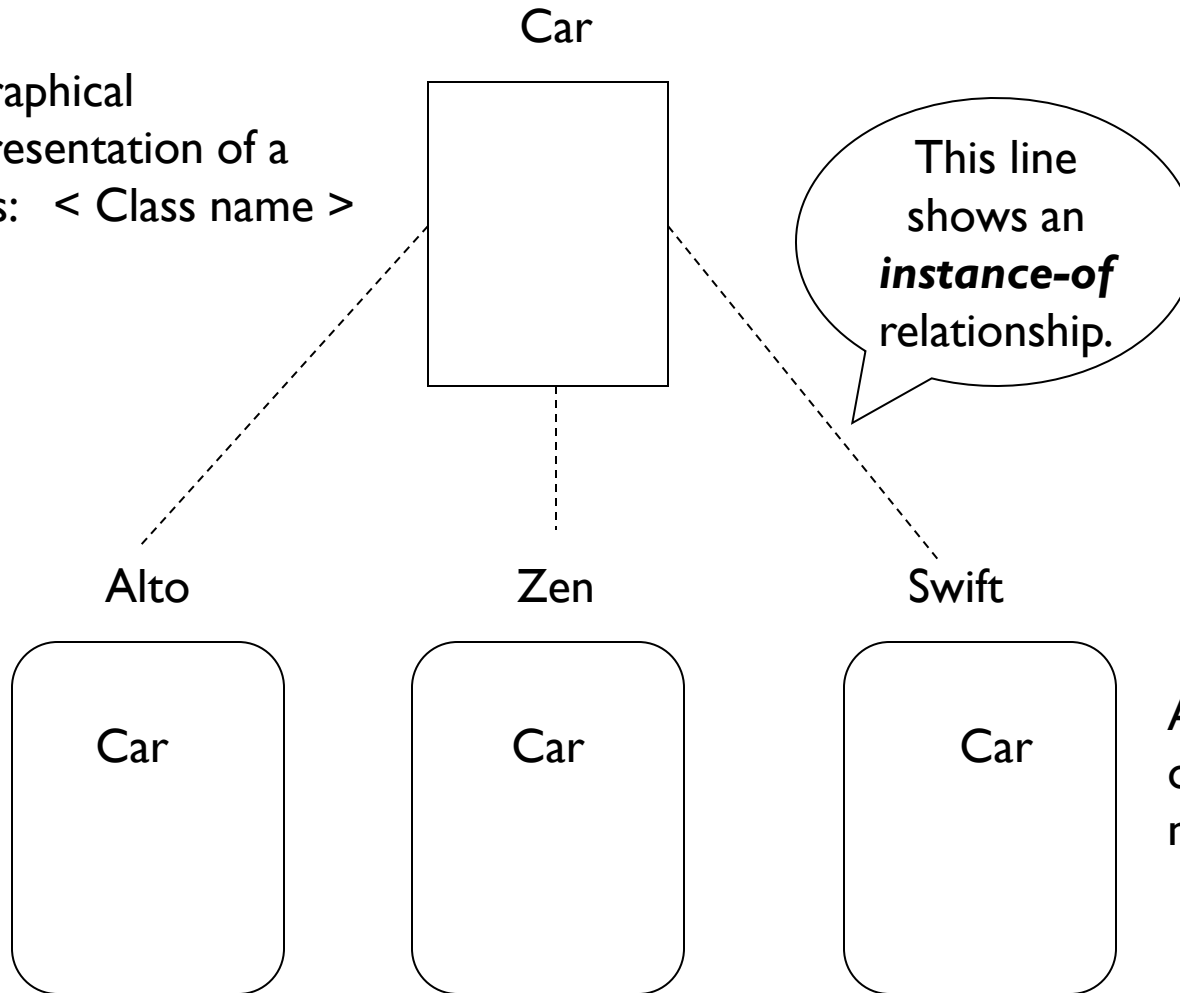
```
Person p1 = new Person ();  
Person p2 = new Person (15);  
Person p3 = new Person ("A");
```

```
class Person {  
    Person(int a) {  
        age = a;  
    }  
  
    Person(char n) {  
        name = n;  
    }  
    ...  
}
```

Overloading of constructors allows the option of creating a class with alternative initializations.

Class and objects - Example

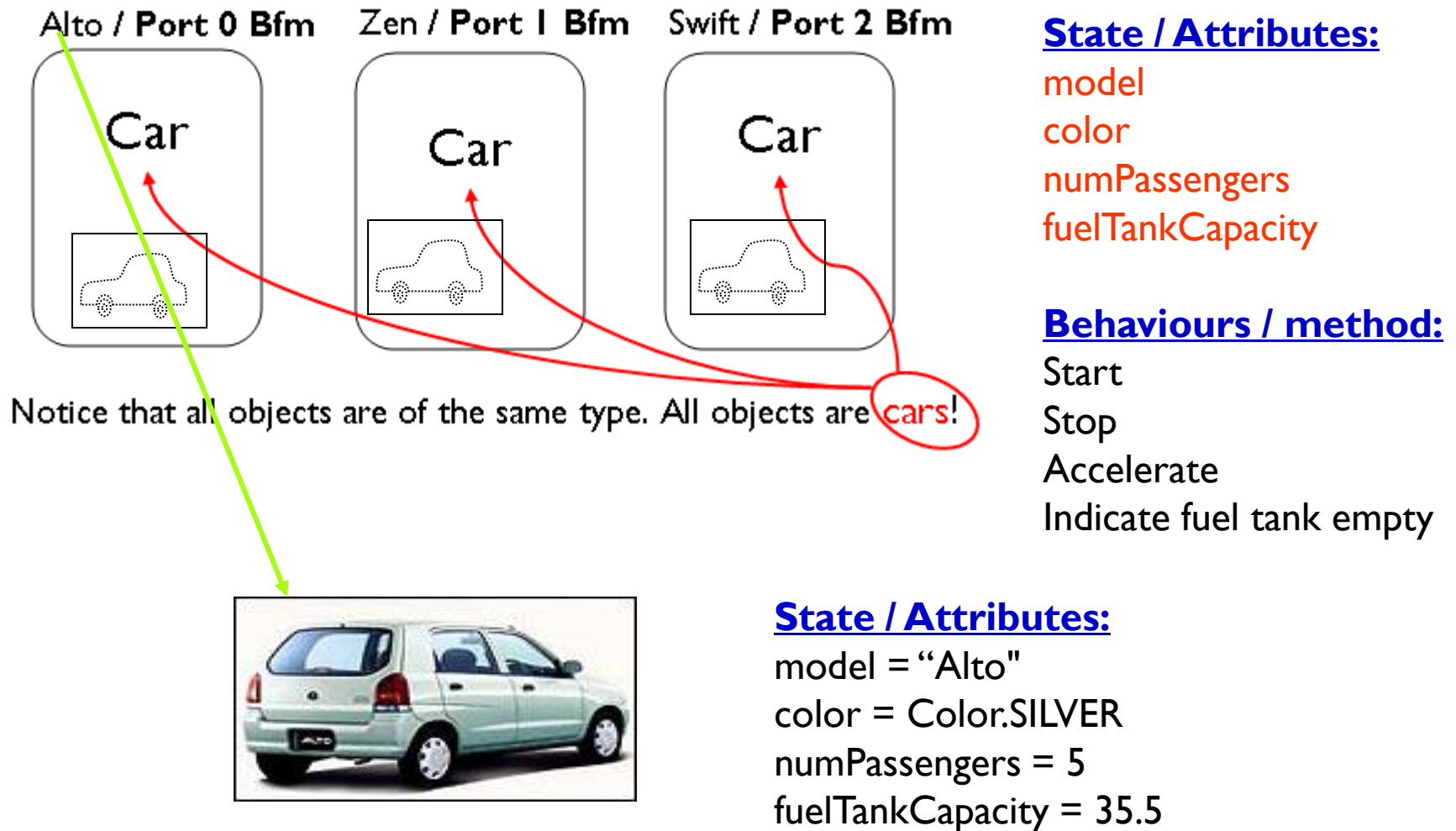
A graphical representation of a class: < Class name >



A graphical representation of an object: < Object name >

Notice that all objects are of same type. All objects are **Cars**!

Example (Contd.).



The Object-Message Passing

Message passing is the act of interacting with one of the object's interface, and invoking the behaviour associated with that interface



Sending “Play” message to the object

Principles of object oriented programming



- Encapsulation & Abstraction
- Inheritance
 - Member Overriding
 - Multiple Inheritance
 - Base class initialization
 - Order of execution
- Polymorphism
 - Overloading
 - Virtual functions
 - Abstract Classes
 - Limitations of Virtual functions

In this module, we discussed:

- Objects and class
- How to identify real world Objects
- Identifying State and Behavior of Objects
- Association of state and behavior with the objects
- Principles of OOP

Abstraction & Encapsulation

Module 3

Objectives



At the end of this module, you will be able to:

- Define Abstraction and Encapsulation
- Illustrate class & objects using code snippet of C++
- Identify the features of constructors and destructors
- Implement overloaded constructors
- Identify the importance of static member of a class

Duration: 4 hrs

- Abstraction arises from recognition of similarities between certain objects, situations, or processes in real world
- Decision to concentrate upon these similarities, and to ignore for the time being the differences
- Abstraction is one of the fundamental ways that we as humans cope with complexity
- An abstraction denotes the essential characteristics of an object that distinguishes it from all other kinds of objects
- Thus provides crisply defined conceptual boundaries, relative to the perspective of the viewer

The user of the walkman needs to abstract the essential details using which he will be able to operate the walkman, and play the music, while at the same time choosing to remain ignorant about the complex internal implementation that is providing him/her the functionality



Encapsulation: Overview

Encapsulation is designed to make an object look like a *black box*: The insides of the box are hidden from view.

Some controls on the outside are the only way that the user can use the Box.

For example:



Play

It means hiding the details of an object's internals (data) from the other parts of a program.

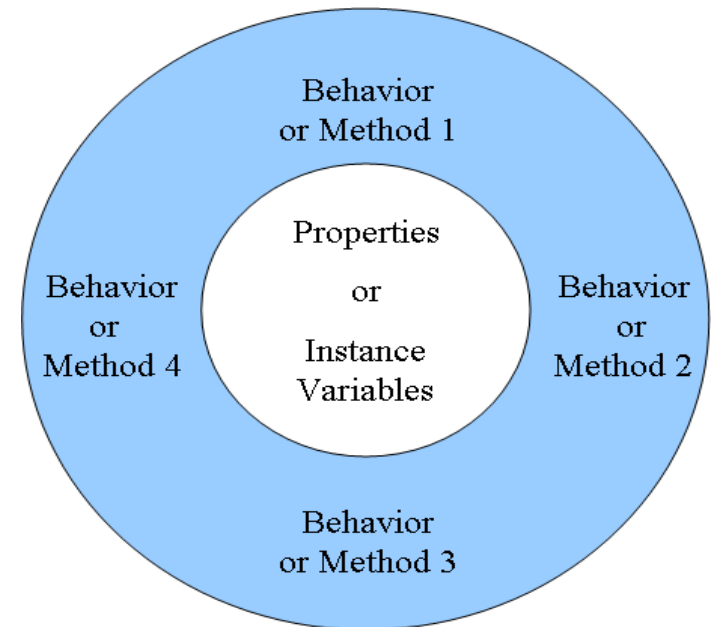
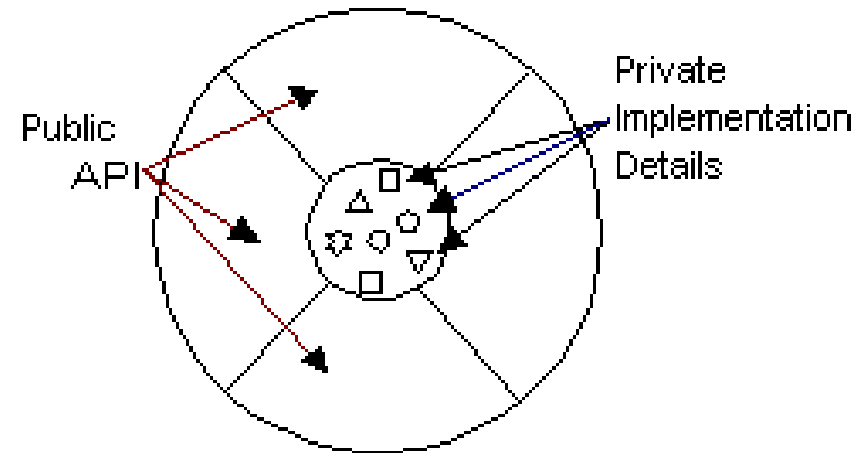
The data (state) of an object is private – it cannot be accessed directly.

The state can only be changed through its behaviour.


The object's data can be used only through its access methods that are carefully written to keep the object consistent and secure.


Encapsulation


- **Abstraction** and encapsulation are complementary concepts.
- Abstraction focuses upon the observable behaviour of an object,
- Encapsulation focuses upon the implementation that gives rise to this behaviour.
- Encapsulation is achieved through information hiding
 - Data is enclosed inside an object
 - Cannot be accessed directly from outside
 - Provides data security




Encapsulation – An Example

 Customer
<i>Attributes</i>
- name : String - rentals : Vector
<i>Operations</i>
+ addRentals(arg : Rental) : void + getName() : String + statement() : String - headerPrint() : String - footerPrint() : String - getInvoice() : String

 Rental
<i>Attributes</i>
- movie : Movie - daysrented : int
<i>Operations</i>
+ getDaysRented() : int + getMovie() : Movie + statement() : String + getFreqUserPoints() : int + getCharges() : double

 Movie
<i>Attributes</i>
+ Children : int + NewRelease : int + Regular : int - priceCode : int - title : String
<i>Operations</i>
+ getPriceCode() : int + setPriceCode(arg : int) : void + getTitle() : String + getChargeCode() : double + getfreqUserPoints() : int

 Price
<i>Attributes</i>
<i>Operations</i>
+ getCharges() : double + getFreqPoints() : int

Illustrative Example – A code snippet

- Encapsulation of type and related operations

```
class point {  
    double x_cord,y_cord;    // private data members  
public:                    // public methods  
    point (int x0,int y0){x=x0;y=y0;} // argument constructor  
    void move (int dx,int dy);        // a method  
    void rotate (double alpha){  
        x_cord= x * cos (alpha) - y * sin (alpha);  
        y_cord = y * cos (alpha) + x * sin (alpha);  
    }  
    int distance (point p);  
}
```

A class is a type : objects are instances

```
void main() {  
    point p1 (10, 20); // call constructor with given arguments  
    point p2;          // call default constructor  
  
    //Methods are functions with an implicit argument  
    p1.move (1, -1);   // special syntax to indicate object  
}
```

Overloading of Constructors



A constructor is basically a function used for initialization – to initialize variables, to allocate memory, and so on

- Special method (s) invoked automatically when an object of the class is declared

```
point (int x0, int y0);
```

```
point ();
```

```
point (double alpha; double r);
```

```
point p1 (10,10), p2; p3 (pi / 4, 2.5);
```

- Constructor name has same name as class name
- Declaration of constructor does not have return type

- Destructors are also functions
 - Their purpose is to de-initialize objects when they are destroyed
 - They are complimentary to constructors
- A destructor is invoked when an object of class goes out of scope, or when memory occupied by it is de-allocated using **delete** operator
- A destructor, like a constructor, is identified as a function that has same name as that of class, but is prefixed with a '~' (tilde)

Visibility Modifiers

- Visibility modifiers help in controlling the visibility of variables and methods within the class
- All parts of a *class* have visibility modifiers or access specifiers
- Usually classes provide 3 access levels to their members (state and behavior):
 - **public**: *The part of class that is visible to all areas other than this class*
 - **protected**: *The part of class that is only visible to subclasses of the class*
 - **private**: *A part of class that is not visible to any other classes*

	public	private
Variables	Violate encapsulation	Enforce encapsulation
Methods	Provide services to users	Support other methods in the class

Object Creation and Initialization



- Every time an object is created, memory is allocated for it. But allocation of memory does not automatically ensure initialization of member data within the object
- Referring to our earlier example of class point, whenever an object of class point is created, we would want to initialize the x_coord and y_coord member data with (0,0), or any specific values

A class is a type : objects are instances

```
void main() {  
    point p1 (10, 20); // call constructor with given arguments  
    point p2;          // call default constructor  
  
    //Methods are functions with an implicit argument  
    p1.move (1, -1);    // special syntax to indicate object  
    }
```

Static members

- Need to have computable attributes for class itself, independent of any specific object; e.g. For creating number of objects
- Static qualifier indicates that unit is unique for the class
`static int num_objects = 0;`
`point () { num_objects++;};` // same for other constructors
- Static data using **class name** or **object name** can be accessed:
`if (point.num_objects != p1.num_objects) error ();`

Note: Static methods can access only static data members in a class

Encapsulation: Hands-on (2 hours)



Purpose

- Identify class(es) along with their Attributes & Methods
- Create multiple objects with different states
- Implement using C++ programming language

In this module, we discussed:

- Abstraction and Encapsulation
- Class & objects using code snippet of C++
- Features of constructors and destructors
- Overloaded constructors
- Static members of a class

Inheritance

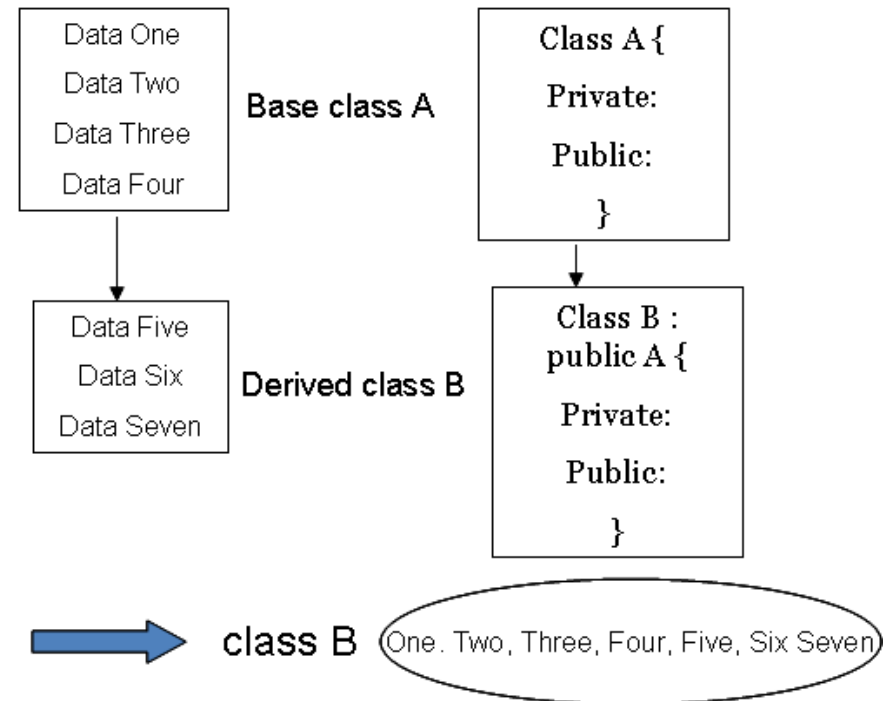
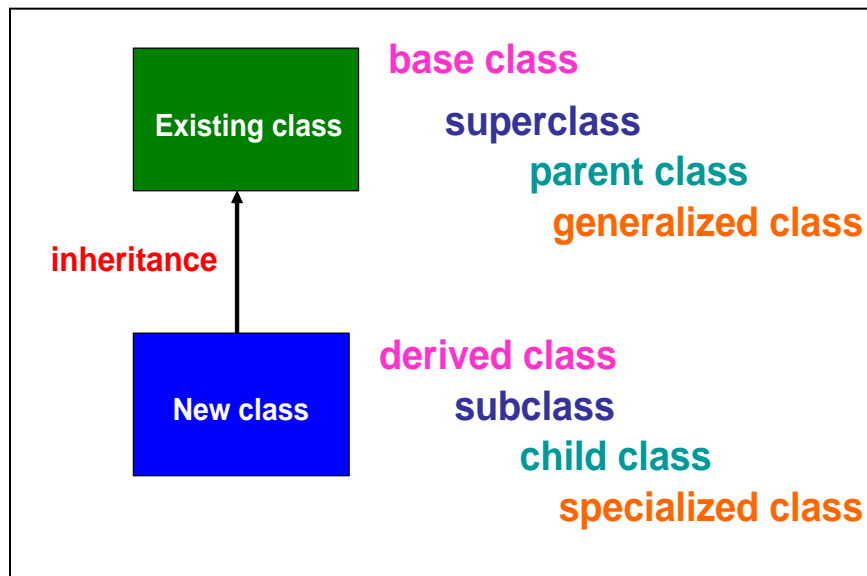
Module 4

At the end of this module , you will be able to

- Identify the need for code reusability in a S/W application
- Determine advantages and disadvantages of Inheritance
- Identify order of constructor execution in Inheritance
- Identify different types of Inheritance

Inheritance: An overview

Inheritance allows a software developer to derive a new class from an existing class.



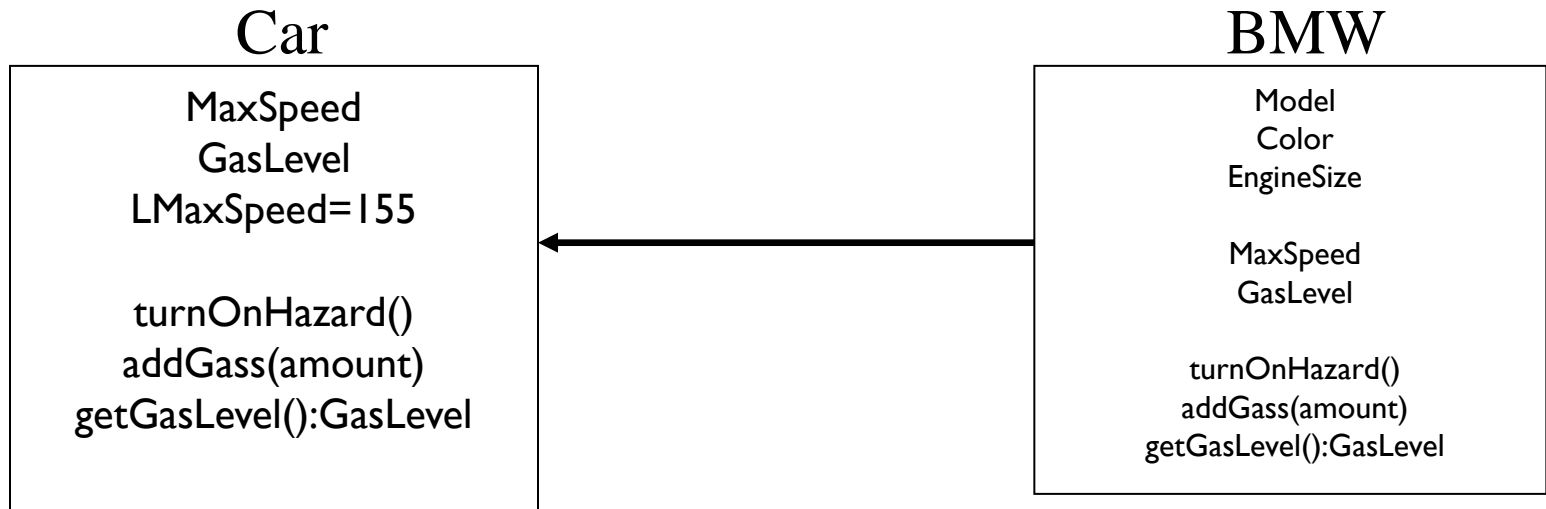
The subclass inherits methods and data defined in super class

Inheritance (Contd.).



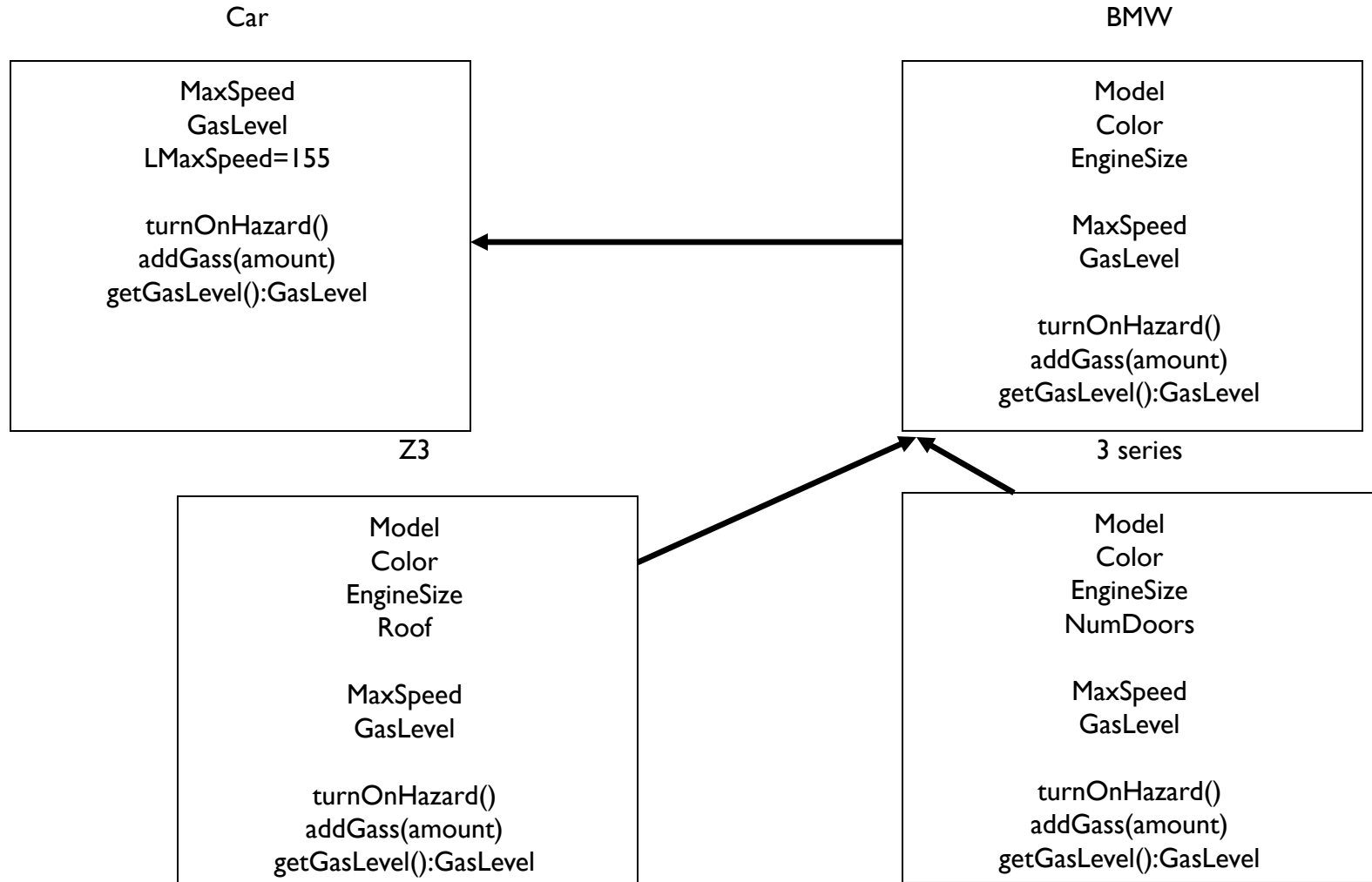
- A class (*SubClass*) can *Inherit* attributes and methods from another class (*SuperClass*)
 - Subclasses supply specialized behavior
 - Provides *Reusability of Code*
 - Avoids Data *Duplication*
- A class which is derived from an existing class is known as a Derived Class/Subclass.
- The class from which other classes are derived is called Base class/Super class
- A subclass not only inherits attributes and behaviors from its super class, but also adds its own unique attributes and behaviors giving it its unique identity

Inheritance – An Example

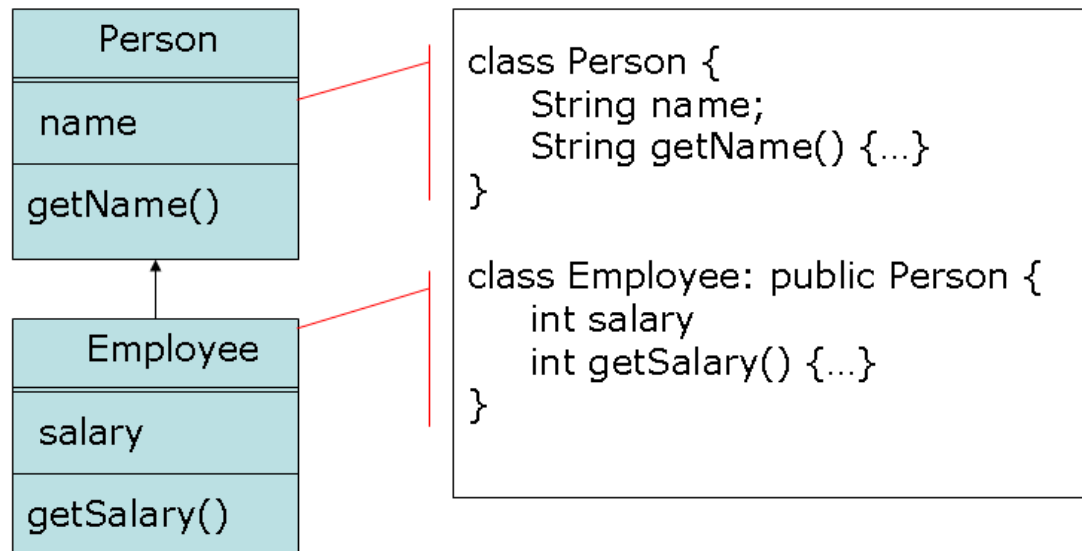
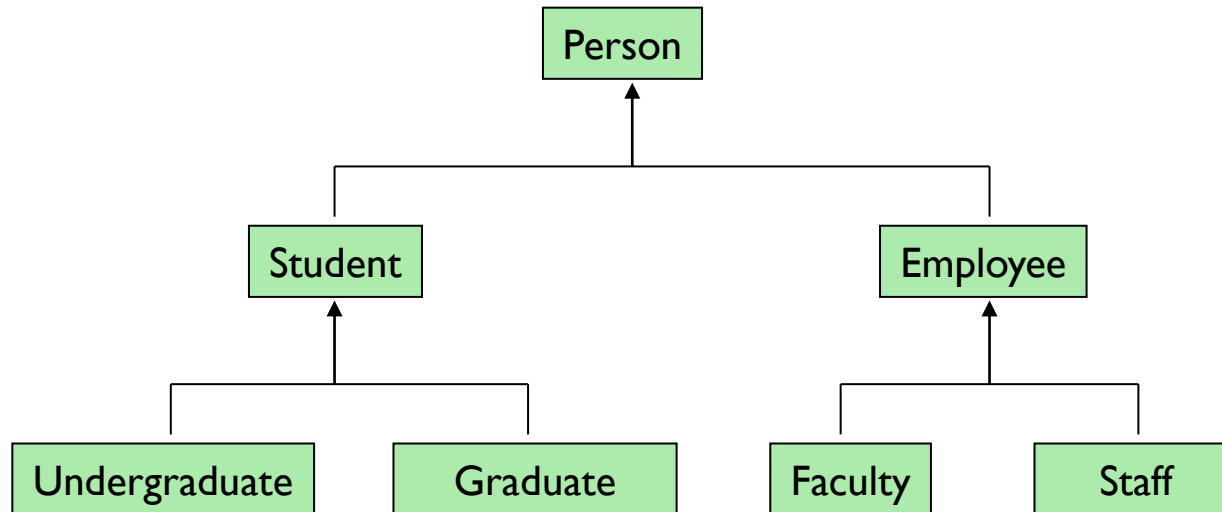


- Now let's define two more new classes. One for the Z3 and another for the 3 series Sedan
 - What might be some of the differences between the two classes?
 - Number of doors (3 or 5)
 - Roof (soft or hardtop)
 - Therefore, we add variables *NumDoors* and *Roof*

Inheritance – An Example (Contd.).



Inheritance - Another Example



Code Syntax for inheritance

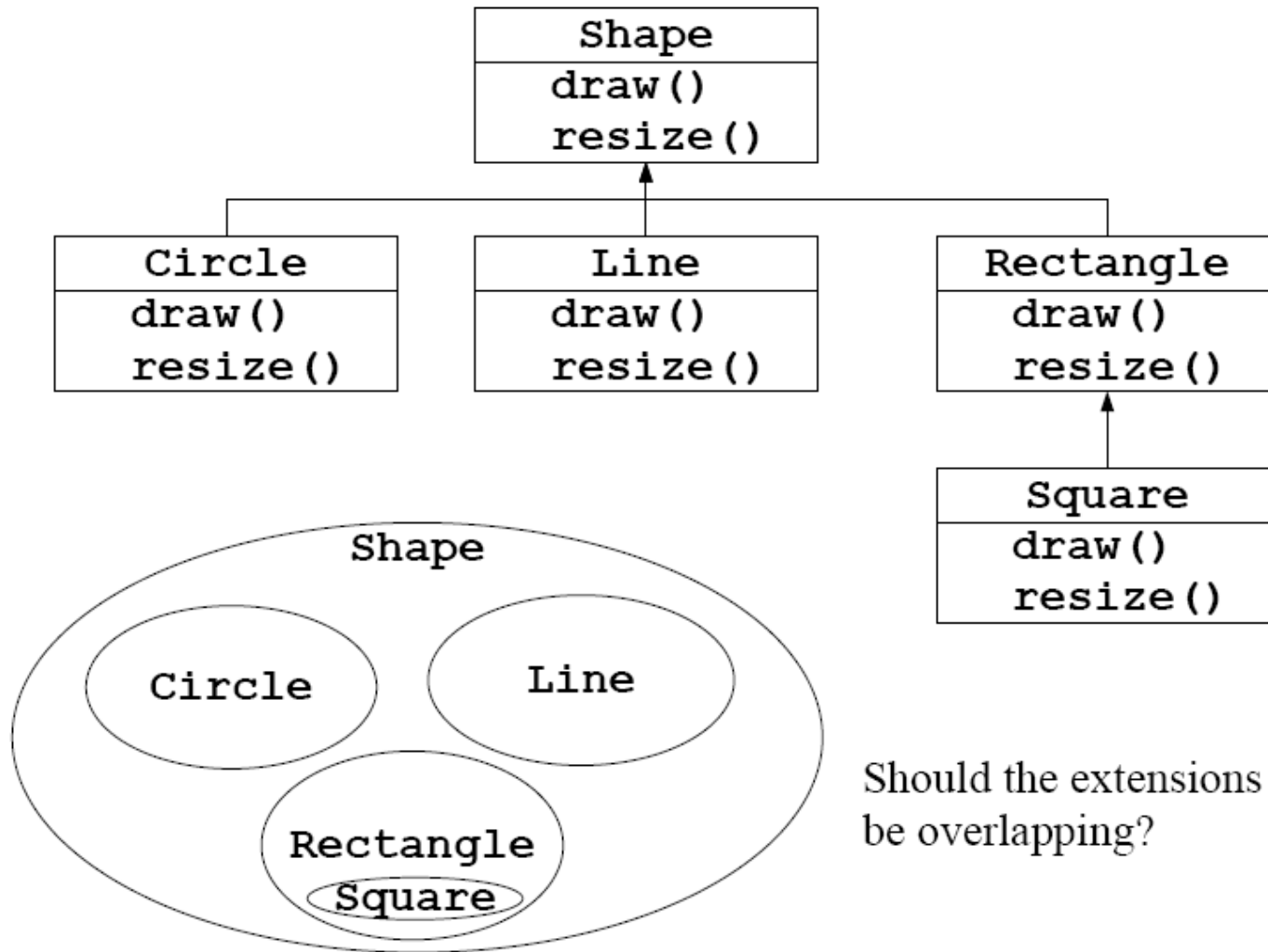


- Class inheritance uses this general form in C++:
- `class derived-class-name : access base-class-name`
- The access status of base class members inside derived class is determined by **access**
- The base class access specifier must either be **public, private, or protected**
- If no access specifier is present, the access specifier is **private** by default

How to Reuse Code?

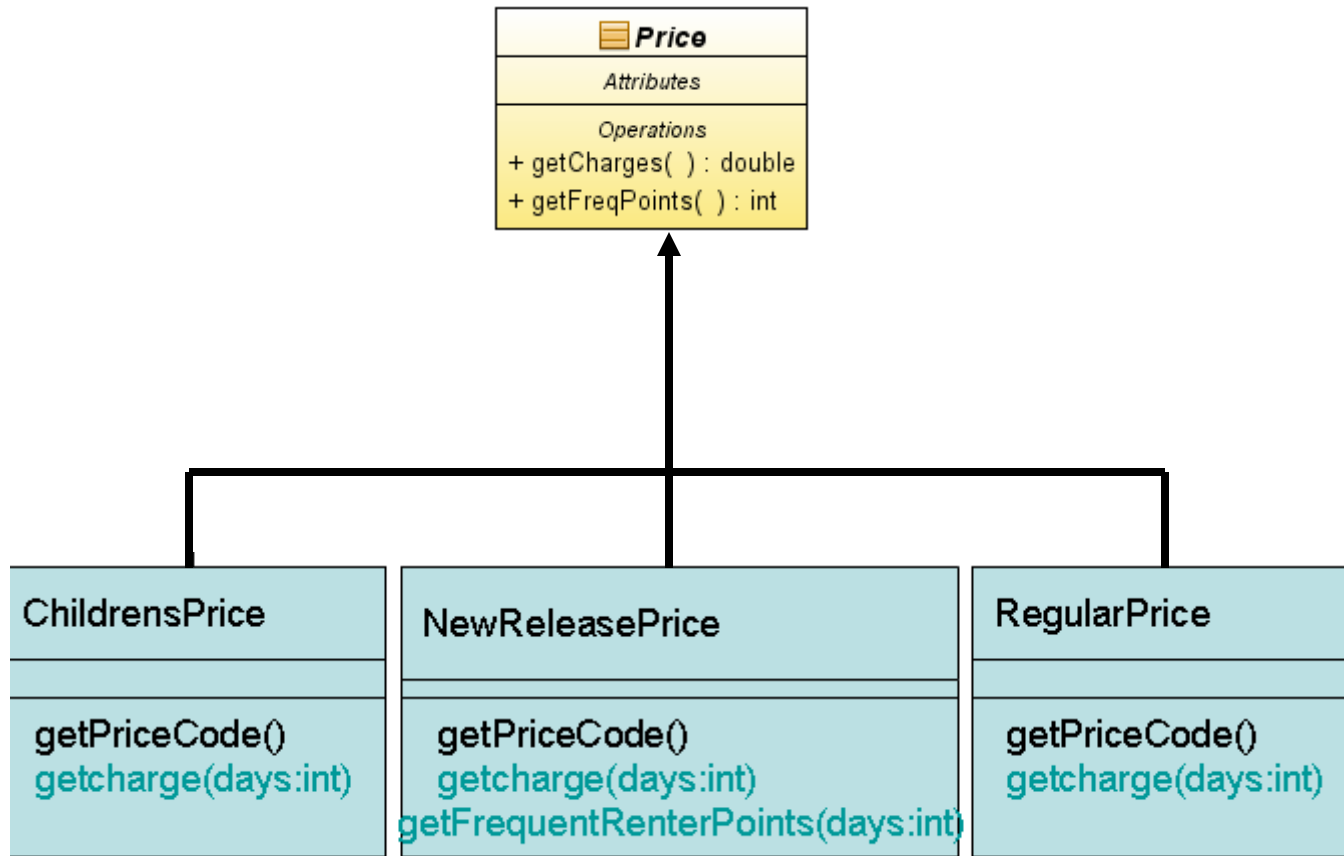
- Write the class completely from scratch (one extreme)
 - What some programmers always want to do!
- Find an existing class that exactly match your requirements (another extreme)
 - The easiest for the programmer!
- Built it from well-tested, well-documented existing classes
 - A very typical reuse, called composition reuse!
- Reuse an existing class with inheritance

Class Generalization / Specialization – Example I



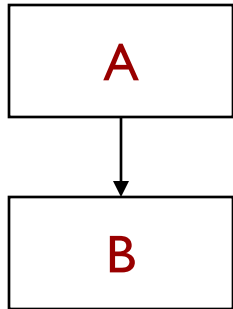
Should the extensions be overlapping?

Class Generalization / Specialization – Example 2

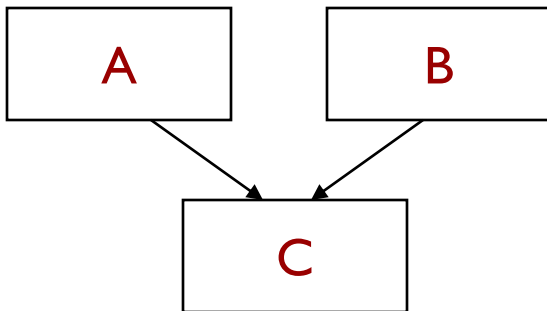


Types of inheritance

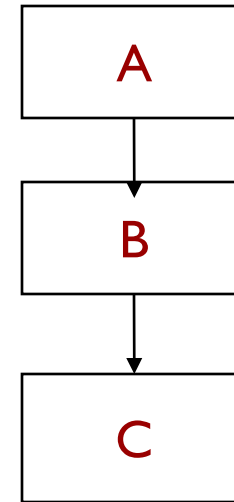
- **Single Inheritance**



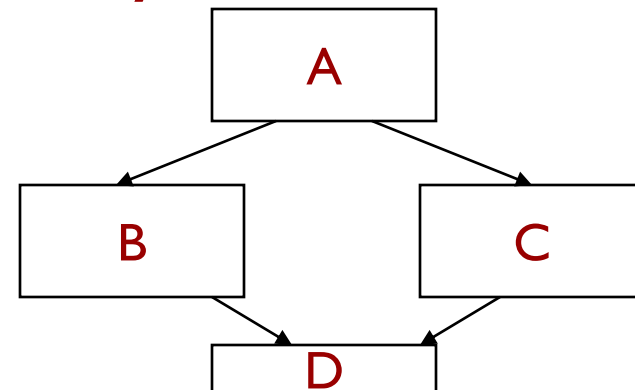
Multiple Inheritance



Multilevel Inheritance



Hybrid Inheritance



Constructors- Execution Order



- When an object of grandchild (last subclass) is created, a general rule that is followed by compiler is:
 - The Virtual base class constructors, in the order of inheritance
 - Non-virtual base class constructors, in the order of inheritance
 - Member objects' constructors (i.e., if an object of one class is declared as a member of another class) in the order of declaration
 - Finally, the constructor of the derived class

Note: Destructors are executed in reverse order

When to avoid/ use Inheritance



- In case you need to use certain methods of class **A** in class **B**, then you will extend class **B** from class **A**, provided class A and class B are strongly related
- If you want to use some methods of class **A** in class B without forcing an inheritance relationship, you determine whether there exists a collaboration between them, and define an association between them
- Since Inheritance induces strong coupling in the design, ensure that we extend a class only if it is strongly related

Inheritance: Hands-on (2 hrs)



Purpose

- To create suitable base class and derived classes to realize Inheritance

In this module, we discussed:

- Need for Code Reusability
- Limitation of Inheritance
- Types of inheritance
- Generalization and Specialization
- Order of execution of constructors in inheritance

Polymorphism

Module 7

Objectives



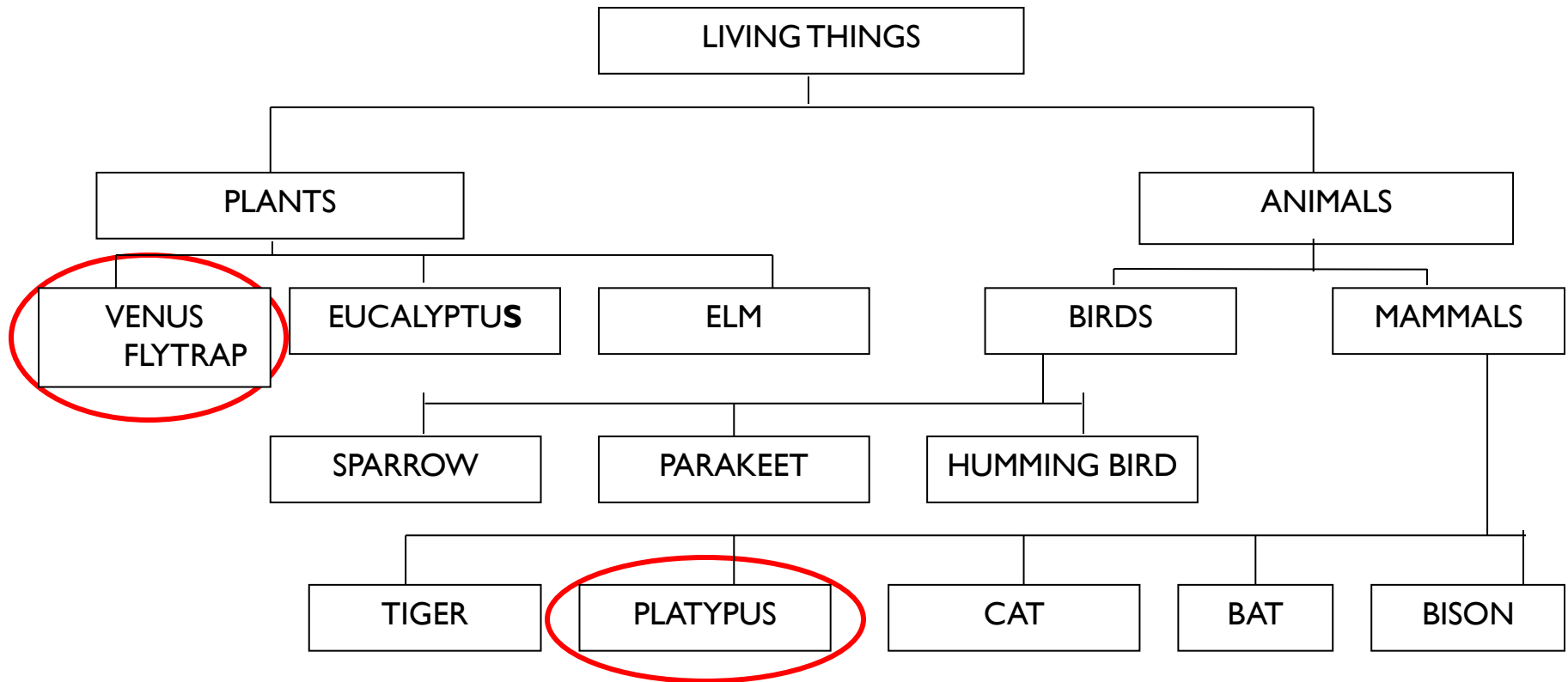
At the end of this module , you will be able to

- Define Polymorphism and identify its major forms
- Explain run time polymorphism with an example
- Distinguish method overloading and overriding

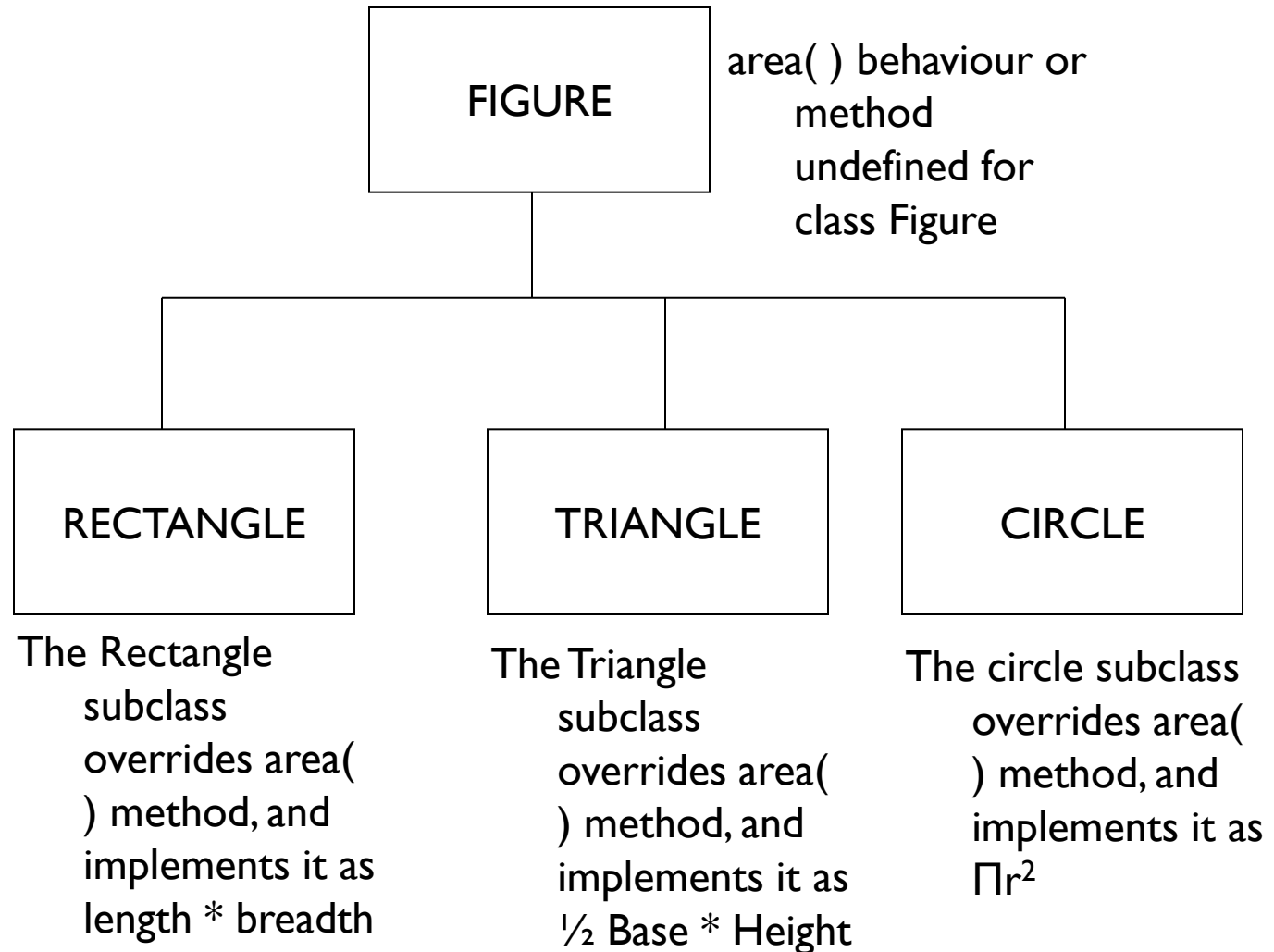
Duration: 4 hrs

- Polymorphism literally means “anything that is capable of existing in multiple forms. (the root words “Poly” and “Morph” are Greek words that stand for “many” and “forms” respectively
- Polymorphism in OO environment is typically associated with overridden behaviors across subclasses in a class hierarchy, each of which has the same name as that in its base class, but chooses to keep its implementation specific to its own needs

Polymorphism in Nature



Polymorphism in Mathematics



Major Forms of Polymorphism



- There are four major forms of polymorphism in object-oriented languages:
 - Overloading (ad hoc polymorphism) -- one name that refers to two or more different implementations
 - Overriding (inclusion polymorphism) -- A child class redefining a method inherited from a parent class
 - The Polymorphic Variable (assignment polymorphism) -- A variable that can hold different types of values during the course of execution. It is called Pure Polymorphism when a polymorphic variable is used as a parameter
 - Generics (or Templates) -- A way of creating general tools or classes by parameterizing on types

Function Overloading



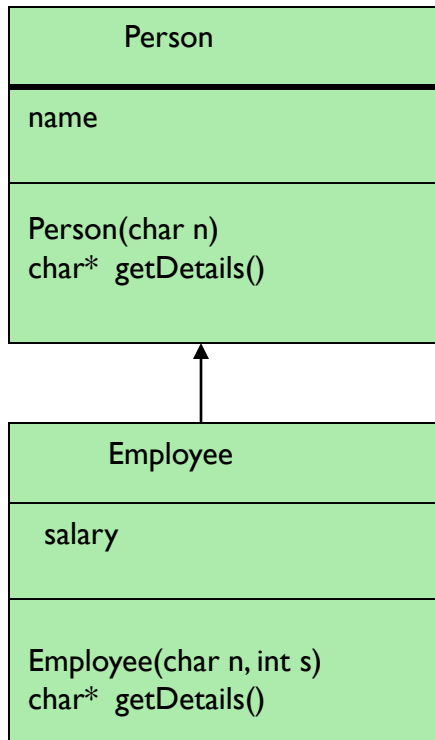
- Two or more functions can share the same name and perform closely-related tasks as long as their parameter declarations are different
- Function overloading improves readability if the functions perform closely related tasks
- A function call first matches the prototype having the same number and type of arguments, and then calls the appropriate function for execution. A best match must be unique
- Example : Overloading of Constructors

Overriding Behaviours



- There are times when a derived class may choose to provide for its own specific implementation of an inherited behavior
- In such a scenario, the derived class chooses to keep the same nomenclature of the behavior as defined in the base class, but chooses to keep the semantics different specific to its own needs
- This is a case where a derived class is said to have overridden a behavior inherited from its base class

Method Overriding



Overridden

Overriding

Output:

Name = Jane
Salary = 20000

```
class Person {
    private char *name;
    public Person(char *n) {
        c_name=new char[strlen(n)];
        Strcpy(c_name,name);
    }
    public char* getDetails() {
        return "Name = " + name;
    }
}

class Employee : public Person {
    private int salary;
    public Employee(int s, char *c) : person (*c){
        salary = s;
    }
    public char* getDetails() {
        return " Salary = " +salary;
    }
}

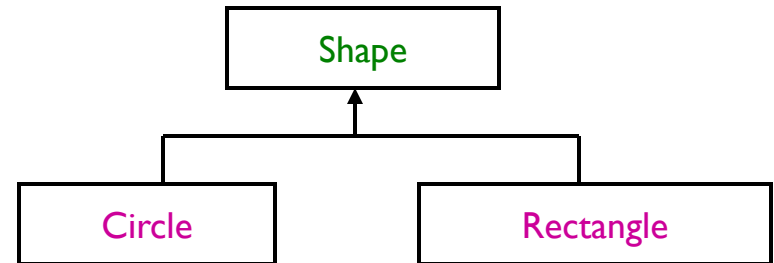
void main() {
    Person p = new Person("Jane");
    Employee e = new Employee("George", 20000);
    cout<< p.getDetails();
    cout<<e.getDetails();
}
```

Polymorphism Example - Shape

This code sample shows polymorphism using “Overriding using inheritance”.

Two subclasses (Circle and Rectangle) are created with Shape as a base class,. Each subclass has

```
class Shape {  
    char c;  
    public: double  
    area();  
    //Method to set the  
    color of shape  
    Public: void  
    setColor (char color)  
    {  
        c = color;  
    } }  
}
```

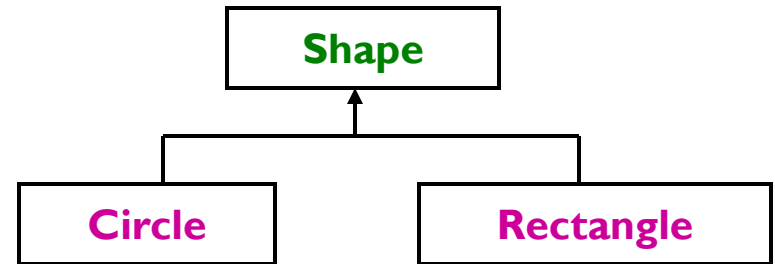


```
class Circle : public Shape {  
    private: double r;  
    double PI  
        =3.1415926535;  
    public: Circle() {  
        r = 2.0;  
    }  
    public double area() {  
        return PI * r * r;  
    } }  
}
```

```
class Rectangle :public Shape {  
    private: double w, h;  
    public Rectangle(){  
        w = 5.0;  
        h = 4.0;  
    }  
    public double area(){  
        return w * h;  
    } }  
}
```

Polymorphism Example – Shape (Contd.).

```
void main() {  
    //set up reference variable for Shape  
    Shape s;  
  
    //create specific objects  
    Circle c = new Circle();  
    Rectangle r = new Rectangle();  
  
    //Now reference each as a Shape  
    s = c;  
    s.setColor("p"); // p is pink  
    Cout<<"Circle color: " <<s.getColor()<< " Area of circle: " << s.area();  
  
    s = r;  
    s.setColor("b"); // b is blue  
    Cout<<"Rectangle color: " << s.getColor() << " Area of rectangle: " << s.area();  
}
```



Which version of area() is called?

Output:

Circle color: p .Area of circle: 12.566370614
Rectangle color: b .Area of rectangle: 20.0

Polymorphism - Example 2

```
class Employee {  
public:  
    void Display() { CalcPaycheck(); }           // non-virtual  
    virtual void CalcPaycheck(); // virtual  
};  
class SalariedEmployee :public Employee {  
public:  
    void Display() { CalcPaycheck();}  
    virtual void CalcPaycheck() {cout << "SalariedEmployee"<<"\n";}  
};
```

```
int main()  
{ Employee * ep = new SalariedEmployee;  
  //...  
  ep->Display();  
  return 0;  
} //salemp.cpp, Result: SalariedEmployee
```

Much of the power of OOP is centered on runtime polymorphism using class inheritance and method overriding:

Polymorphism refers to the ability for objects of different classes to respond differently to the same method call

Polymorphism is implemented using a technique called **late method binding or dynamic method binding**; that is which right method to call is decided during run-time based on the referenced object

- Virtual functions and polymorphism
- Virtual Function
 - A non-static member function prefixed by the virtual specifier.
 - It tells the compiler to generate code that selects the appropriate version of this function at run time
- Abstract and Concrete classes
- Provides ability to manipulate instances of derived class via a set of operations that is defined in their base class
- Each derived class can implement operations that are defined in the base class differently, while preserving a common class interface that is provided by base class

Runtime Polymorphism – Example I

```
#include <iostream.h>
class Employee {
public:
    long GetDepartment() const{return deptNum;};
    long GetId() const{return id;};
    void SetDepartment( long deptId ){};
    void SetId(){};
    virtual void CalcPaycheck() =0;
    virtual void Input(){};
private:
    long id;
    long deptNum;
};
```

```
class SalariedEmployee :public Employee {
public:    Void CalcPaycheck(){
            cout<<"SalariedEmployee"<<endl; };

class HourlyEmployee :public Employee {
public: void CalcPaycheck(){
            cout<<"HourlyEmployee"<<endl;};
        void ProcessAnyEmployee( Employee & er
        ){
            long anId = er.GetId(); // non-virtual
            er.CalcPaycheck();      // virtual
        }    };
```

```
void main()
{
    SalariedEmployee S;
    HourlyEmployee H;
    ProcessAnyEmployee( S );
    ProcessAnyEmployee( H );
}
//calcpay.cpp
```

- Dynamic binding is a process of invoking a specific sequence of code during runtime
- This is referred as Runtime polymorphism too
- For example: When multiple classes contain different implementations of the same method, say `display()`. If the class of an object Employee is not known at compile-time, then when `display()` is called, the program must decide at runtime which implementation of `display()` to invoke, based on the runtime type of object.

In Conventional Programs

You should write:

if (Shape.type is circle)	then	DrawCircle(Shape);
else if (Shape.type is rectangle)	then	DrawRectangle(Shape);
else if (Shape.type is point)	then	DrawPoint(Shape);
else if (Shape.type is line)	then	DrawLine(Shape);

Using Polymorphism

- You need only to write: Shape.Draw()
- In C++, it is a pointer to Shape, then you can set a pointer to the other shapes to the pointer Shape, at last ,you can use Shape->Draw();
- Actually, the pointer in C++ is a type that can denote anything, such as, void * vp;

- An abstract class is a class that can only be a base class for other classes
- Abstract classes represent concepts for which objects cannot exist
- A class that has no instances is an abstract class
- Concrete Classes are used to instantiate objects
- An Abstract Class
 - In C++, a class that has one or more virtual functions is an abstract class
 - An abstract class either contains or inherits at least one pure virtual function.
 - A pure virtual function is a virtual function that contains a pure-specifier, designated by the “=0”

Abstract Classes: Example

```
#include "iostream.h"
```

```
class base{  
public:  
    virtual void display(){  
        cout<<"I am in base class"<<endl;  
    }  
};
```

```
class derived1:public base{  
public:  
    void display(){  
        cout<<"I am in derived1 class"<<endl;  
    }  
};
```

```
class derived2:public base{  
public:  
    void display(){  
        cout<<"I am in derived2 class"<<endl;  
    }  
};
```

```
void main()  
{  
    base *xyz;  
  
    xyz=new base;  
    xyz->display();  
  
    xyz=new derived1;  
    xyz->display();  
  
    xyz=new derived2;  
    xyz->display();  
}
```

Polymorphism: Hands-on (2 hrs)



Purpose

- To realize runtime polymorphism (dynamic binding)

- Object-Oriented design involves decisions on:
 - Decomposition of a problem
 - Representation of physical concepts as classes
 - Relationships between classes
 - Class interfaces: member functions and operators
 - Concrete implementations of interfaces

- Unified Modeling Language (UML) assists OO design and analysis
 - Class diagrams: classes, interfaces and class relationships
 - Object diagrams: a particular object structure at run-time
 - Interaction diagrams: flow of requests between classes

In this module, we discussed

- Concept of Overloading and Overriding
- Need for Polymorphism with respect to inheritance

- Booch, Grady. *Object Oriented Analysis & Design with Applications*. Ed 3. New Delhi: Pearson, 2007.
- Eckel, Bruce. *Thinking in C++: Introduction to Standard C++*. Vol I. Ed 2. New Delhi: Prentice Hall, 2000.



Thank you