Object Oriented Concepts
Course Objective

- To create an awareness for the various ways of handling software complexity
- To explain the concepts of Object Oriented Technology
- To understand the different features of Object Oriented Technology
- To understand the difference between Structured Programming and Object Oriented Programming

At the end of this course, you will

- understand the inherent complexity involved in software systems and will gain the knowledge of handling such complex software systems

- be able to differentiate between the two programming styles – Structured Programming and Object Oriented Programming

- learn the different features of Object Oriented Technology
References

Session Plan

• How to manage software complexity?

• Why Object Orientation?

• Features of Object Oriented Technology

Manage Software Complexity

• Different approaches for solving a problem which is complex in nature

Why Object Orientation to solve a complex problem?

Differentiate between Structured programming and Object Oriented Programming approach

To discuss the different features of Object Oriented Technology
Structured Programming

• The structured programming paradigm is:
  • Decide which procedure you want
  • Use the best algorithm you can find
• Here the focus is on the algorithm required to perform the desired computation
• In this style of programming, importance is given to procedure (logic) and not to the data on which these procedures operate

Key points in structured programming

• Focus is on process rather than on data
• It is best suited for a simple solution
• Design approach is “Top-Down” where the entire solution is divided into smaller units (Functions and procedures)
• All these smaller units need to work on a data item to return the result
• For this reason the data items used are Global
• Modules are tightly coupled because of which the same module cannot be reused in another scenario.

• Coupling:
  • Coupling refers to the manner and degree of interdependence between software modules. (IEEE)
  • Coupling applies to any relationship between software components.
  • Can be defined as mutual dependence of methods. Low coupling is good for design.

• What is the problem if the modules are tightly coupled?
  • If the modules are tightly coupled, it makes the system complex as the module is tough to understand. Also it is hard to change or correct such a module by itself if it is highly interrelated with another module.
Structured Programming (Contd…)

• Modules are tightly coupled

• They work on global data items

• Structured programming is often associated with a "top-down" approach to design

Structured Programming becomes difficult to manage as the complexity increases because they work on global data items.

• Procedural method for developing information systems
  • works fine for automating routine processes like payroll checks.
  • works well in cases where data and applications are separate.
  • works well in cases where data comes in the start of the program, flows through a number of predefined procedures, and exits at the end.

• Structured Programming fails to address the complexities and needs of interactive environments where the flow control is not linear. The program flow dictates the flow of control to the user.

• Object Oriented Technology promises to ease the software complexity by providing a fundamental change to the way information systems are developed.

• Top-Down Approach:
  • Programmer should break larger pieces of code into shorter subroutines that are small enough to be understood easily.
Object Oriented Programming

- Uses the bottom-up approach
- Data takes precedence over the process
- Promotes Reusability
- Exploits Commonality
- Reduces Complexity
- Facilitates maintenance

Complexity - > It is proportional to the sum of the things which we need to consider + their interaction

Why is Complexity so difficult to handle ?

- Example 1: There is a famous rule saying “7 ± 2”. This states that a human being cannot properly calculate more than 7 plus or minus two things “Simultaneously”. This is a limitation of the human brain. But we never try to get shy away from such scenarios. We learn and try to handle them.

- Example 2: Sometimes when we try to remember a set of numbers (data), we correlate these numbers to an event so that it becomes easier for us to recall the same. Let us take an example. If some one has to remember a set of data (2,3,4,1,9,7,2) , we would like to remember this as “Sajjad (My friend) born on 23 April 1972”. This is easy to remember. But when the set of data increases it becomes difficult to handle this.

Why Object oriented Programming ?

- Object oriented technology offers a new model where data and processes are integrated.
- Data and processes are packaged together to form an object.
- The object contains all the information and the data which is required to provide the functionality for which it is designed.

Advantages of Object Oriented Programming:

- One of the principal advantages of object-oriented programming techniques over Structured Programming techniques is that it allows programmers to create modules that do not need to be changed when a new type of object is added. A programmer can simply create a new object that inherits many of its features from the existing objects.
- The user dictates the flow of control to the application, thereby being very user-friendly and providing flexibility.
- It’s considered to be more suitable for representing real world objects than Structured programming.
Structured Methods
• Structured methods: An early design strategy with an emphasis on hierarchical decomposition
• Finite state machines, data flow diagrams, entity relationship diagrams
• Works great for smaller systems (<50,000 LOC)
• Fails for larger systems
  • Complex behavioral relationships
  • Complex data-behavior interaction
  • Increased coupling

The Object-Oriented Paradigm
• Collects the data with their associated functions
• Collections communicate via messages

Message Communication:
Object oriented programming consists of a set of objects that communicate with each other. The process of programming in an object oriented language therefore involves the following basic steps:
1. Creating classes that defines objects and their behavior.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending or receiving messages. The concept of message passing makes it easier to talk about building systems that directly model or simulate the real-world. A message for an object is a request for execution of a procedure/function. The receiving object will generate the desired result according to the procedure/function call.
Features of Object Oriented Technology

- **Abstraction**
  - Class
  - Object

- **Encapsulation**
  - Access Specifiers
  - Data Hiding

- **Inheritance**

- **Polymorphism**
  - Static and Dynamic Polymorphism

- **Abstraction** - It denotes the essential characteristics of an object that distinguish it from all other kind of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer. It focuses on observable behavior of an object.
  - **Class** - It is a set of objects that share a common structure and a common behavior.
    - Data Member
    - Member Methods - denotes the service that a class offers to its client.
  - **Object** - Object is an instances of a class. Thus an object is an entity that has attributes and provide certain operations that are defined for the particular object.

- **Encapsulation** - Focuses upon the implementation that gives rise to the observable behavior.
  - Access Specifiers
  - Data Hiding

- **Inheritance** - Opens the possibility of extending and adapting object description and implementation without changing their source code.
Abstraction

• **The art of being wise is the art of knowing what to overlook** - William James

• Focuses upon the essential elements and characteristics of some object, relative to the perspective of the viewer

• In Object Oriented Programming, we try to handle complexity by using the concepts of class and object

William James’ definition in the slide above points at handling complexity. We need to use a technique that will reduce the amount of facts we have to deal with simultaneously. Comprehension is not automatic. The time we need to comprehend something is inversely proportional to the number of things we are presented with and to the relevance of those items.

• **Example 1:**
  • Physician find the facts about the patient
    • Normally when a patient goes for a general check-up, the Physician on looking at the previous history of the patient, will understand the situation much faster than if he has to go for a full investigation.
    • The physician is trying to avoid the irrelevant data items so that he can come to the root cause as soon as possible.

• **Example 2:**
  • Take an entity as **BOOK**. Let us try to find out the different characteristics of the same entity from the perspective of the viewer.
  • Let us take 2 cases where the same entity BOOK can be viewed differently:
    • Library System
      • In this case we will be focusing on Access Number, Book Name, Author Name
    • Shopkeeper
      • In this case we will be focusing on Item Number, Item Name, Price, Quantity On Hand.
What is an Object?

- An object
  - Is an unique, identifiable, self-contained entity that possesses operations and contains attributes
  - Possesses all the know-how and information it needs to perform the services for which it was designed
  - Is a "black box" which receives and sends messages

As procedures are used to build structured program, objects are the building blocks of object oriented programs

- A primary rule of object-oriented programming is - as the user of an object, **you would never need to know what is there inside the object!**
- These characteristics represent a pure approach to object-oriented programming:
  - Every object contain some member variables and member methods which work upon the member variable.
  - A program is collection of objects, which needs to interact among them to do a process. The interaction of objects is also called as message passing.
  - Every object has a type as objects instantiated from a class, Here class is considered as a type.
- Objects have **state, behavior, and identity**
- **Every object:**
  - **Contains data:** The data stores information that describes the state of the object.
  - **Has a set of defined behavior.** This behavior consist of all the things that the object "knows" how to do. These are the methods present inside the object.
  - **Has an individual identity.** Each object is different from the other object even if they are instantiated from the same class.
What is a Class?

- A Class
  - is a software template that defines the methods and variables to be included in a particular kind of Object.
  - Is a blueprint used to create objects. As it is a **blue print**, at run time it will not occupy any memory.

- Examples:
  - Animal, Human being, Automobiles

- We never actually write the code for an object: what you write is the classes that is used to make objects.

- Classes increase the efficiency and power of the object by:
  - Classifying objects
  - Relating objects to one another
  - Providing a mechanism to define and manage objects
Class Contains ..

- Member Data
  - Variables defined inside a class
  - Normally Member data are hidden

- Member Methods
  - Functions defined inside the class
  - Member methods are public in nature and accessible from outside the class

- Every object belongs to (is an instance of) a class.

- An object may have fields, or variables
  - The class describes those fields with the help of member data.

- An object may have methods
  - The class describes those methods with the help of member methods.
Encapsulation

• A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module - IEEE

• Principle of binding processes (Member Method) and data (Member Data) to form an integrated unit is the essence of encapsulation

• Implemented by using the concept of Class

- Encapsulation is “a technique that separates the external aspects of an object from the internal implementation.
- A "capsule" or module, packages together functions and the related data, and protects its data from other modules or outside functions.
- Best example here can be : the difference between “Capsule” and “Tablet”. Here we can see that “Tablet” is exposed whereas “Capsule” is within a package.

- The main objective of Object oriented programming is to
  • hide the data item
  • Accessibility of data item with a class is only possible through member methods.
- Abstraction & Encapsulation :
  • Abstraction tells us what external face we should present to the world where as
  • Encapsulation ensures that the implementation of the interface doesn’t leak out to the outside world.
- Encapsulation hides the details of how we implement the state behind the behavior which we present to the outside world.
Access Specifiers

• Used for accessibility of data member and member methods from within and outside the class boundary

• Example:
  • Private
  • Public
  • Protected
  • Note: Some OOP language uses more access specifiers also.
    (For e.g. Java supports Friendly)

• This concept allows the data items to be protected from outside.

• Data member should be allowed to be accessed only by the member methods.

• The derived class inherits all the member methods of the base class.
Data Hiding

- Process of hiding the members from outside the class
- Implemented by the concept of “Private” access specifiers
- Can be accessed only by the member methods of that class where it is defined
- Data hiding is an important feature of OO programming which allows the data member to be kept safe

- This is also called as **information hiding**.

- **IEEE defines Information Hiding as :**
  
  A software development technique in which each module’s interfaces reveal as little as possible about the module’s inner working and other modules are prevented from using information about the module that is not in the module’s interface specification.
Inheritance

- Ability to compose new abstraction from existing one
- Implements the concept of **Re-usability**
- Classes are arranged in a tree like structure called a hierarchy
- Base class:
  - the class providing the implementation
- Derived class:
  - the class inheriting the implementation

- Inheritance also promotes **reuse**. You don't have to start from scratch when you write a new program. You can simply reuse an existing repertoire of classes that have behaviors similar to what you need in the new program.

- Base class is also called Super Class and Derived class is called Child class.

- Derived class can also override the inherited method

- By using the concept of inheritance, it is possible to create a new class from an existing one and add new features to it. Thus inheritance provides a mechanism for class level reusability.
Inheritance: Example

- “Shape” is the root class
- “Polygon” is a derived class from “Shape” class
- “Polygon” is the super class for “Triangle”,
- Generalization and Specialization are associated with the concept of Inheritance

- In a class hierarchy, every class has a superclass except for the class at the top of the hierarchy; this class is called the root class.

- In our example, Shape is the root of the hierarchy. From the other end, the classes that don’t have any subclasses, like Rectangle and Isosceles are called leaf classes.
Generalization and Specialization

• Consider the two classes in the diagram (Person and Student)

• “Student” class inherits the properties from “Person” class

• “Person” is a generalization of “Student”

• “Student” is a specialization of “Person”

• **Specialization** is the process as we go down the class hierarchy

  • of maximizing the differences between members of an entity by identifying their distinguishing characteristics
  • of defining subclasses that are more refined than superclass – its specialized

• **Generalization** is the process as we go up in the class hierarchy

  • of minimizing the differences between entities by identifying their common characteristics
  • of defining Superclass which contains common characteristics of its subclasses – its generalized
Multiple and Multilevel Inheritance

- **Single inheritance** is having common characteristics with only one superclass.
  - Ex: Class “Quadrilateral”

- **Multiple inheritance** means that a class has common characteristics with more than one superclass.
  - Ex: Class “Square”

- **Multilevel inheritance** talks about level of inheritance hierarchy.
  - Ex: 4th Level

- **Single inheritance**, where each class (except the root) has one and only one superclass.

- **Multiple inheritance**, where a class inherits from more than one superclass.

- **Multilevel inheritance**, where class hierarchy goes for more than one level. The level of hierarchy in the example shown in the slide above is 4.
Advantages of Inheritance

• Software reusability

• Design and code sharing

• Software components

• Rapid prototyping

• All benefits apply to design, development and maintenance
Abstract Class

- A class that contains one or more abstract methods (methods which are not complete in terms of implementation)

- Other classes can extend them and make them concrete by implementing the abstract methods

Abstract Class:

At times some of the base classes in our hierarchies do not represent anything concrete enough to instantiate an object in their own rights. Such a class only exists as a ‘holder’ for the shared (inherited) attributes and methods of derived classes and is known as an ‘abstract’ class, because it does not represent a concrete type of object.

Example of Abstract Class:

When we try to think of designing a class for animal, we know all animals exhibit attributes called color and speed. But when we try to initialize these two attributes in methods, then we don't know with what value to initialize them. If it is not a specialized class, it is very difficult to initialize these attributes. In C++ we can define a Abstract class as follows:

class animal
{
    Private:
    int speed;
    char color[20];
    public:
    virtual void init_speed()=0;
    virtual void init_color()=0;
}; // Class definition ends here
Abstract Class (contd..)

- Instantiation of an abstract class is not possible

- Pointer of an abstract class is possible

In C++, if one class contains at least one **pure virtual function** (a *virtual function is a function whose function body is equated to 0*) it is called as abstract class.

Let us look at the body of the two methods defined above. We cannot initialize the variables called speed and color unless until it is not a specialized class and they are declared as virtual as well as equated to 0.

When you say virtual somefunc(...) = 0; you're telling the compiler that make the derived class implement this method and provide a sensible behavior. Once you declare a function as pure, you can't create an instance of the class.

**In Java**, you make a method abstract by using the abstract keyword. You prefix the keyword before the method definition to say the method is abstract. Like this:

```java
abstract class Shape
{
    abstract void draw();
}
```

The Java compiler also makes sure you declare the class abstract too, so that you don't end doing something like this by mistake.
Polymorphism

- Is the ability of different objects to respond in their own unique way to the same message
- Implemented by **overloading** and **overriding**
- Possible because of Encapsulation and Inheritance
- Example of a class hierarchy where vehicles is the base class and there are some classes which are derived from vehicles. (Car, Plane, Bicycle)
- Each of the classes need a functionality called “Stop”

Polymorphism !!

- Polymorphism allows us to send identical messages to dissimilar but related objects and achieve identical actions while letting the software system to determine how to achieve the required action for the given object.

- The sender of a message does not need to know the receiver's class.

- It serves the purpose of structuring systems around **WHAT** is to be done, and allows details about **HOW** classes and methods are implemented to be hidden.

- It allows a general class to specify methods that will be common to all of its derivatives, while allowing subclasses to define the specific implementation of some or all of those methods.
Polymorphism: Example

- Explanation of the above Example:
  - All vehicles offered an operation "stop".
  - Implementation of the "stop" method will be different for different classes.
  - At runtime user will invoke a method "stop".
  - The effect is that upon receiving the message the object knows to which class it belongs to, and thus executes the "stop" method for its own class.

- Message is the same called "stop" but which method is going to get invoked depends upon which type of class the object is sending the message.

- Message is the same but the execution might vary depending upon the derived class implementation.
Polymorphism

• Different Object Oriented Languages use different concepts of implementing polymorphism
  • Function Overloading
  • Operator Overloading
  • Function Overriding

• C++ supports all the above features where as java does not support Operator Overloading.
Function/Method Overloading

- Ability to define two or more methods with the **same names** and **different signatures**
- Signature means number of arguments, type of arguments and sequence of arguments
- Ex:
  1. `void disp(int ix);`
  2. `void disp(int ix, int iy);`
  3. `void disp(int x, char cy);`
  4. `void disp(char cx, int iy);`

- **Method Overloading**
  - When a Class is having more than one method with the same name but of different signature.
  - Function prototype 1 and 2 differ in number of arguments
  - Function prototype 2 and 3 differ in type of arguments
  - Function prototype 3 and 4 differ in sequence of arguments

- Definition Given by IEEE:
  - *To assign an operator, identifier or literal more than one meaning, depending upon the data types associated with it at any given time during program execution.*
Operator Overloading

• Operator overloading in the language gives a new meaning for the operands of a specific class
• Operator overloading provides a way to define and use operators such as +, -, *, /, and [] for user defined types such as classes
• Some of the OOP languages support this

• Why overload operators?
  • It's common to define a meaning for an existing operator for objects of a new class.
  • We want to use the operators for user defined data types (user defined class) also.
  • C++ supports overloading of operators where as java does not allow this.
Overriding

- Process of defining the methods with the same name and signature in derived class which already exists in the Base/Parent class
- Process of re-defining the methods in the derived class
- Methods must have argument lists with the identical type and order as the superclass

- Subclasses can alter or override information inherited from parent classes for exceptional cases
  - All mammals give birth to young ones
  - Some mammals lay eggs
- The ability of a subclass to override a method in its superclass allows a class to inherit from a superclass whose behavior is “close enough” and then override methods as needed.
- When you override a method of a superclass you should honor the intended behavior of the method.
Example in C++:

class animal
{
    protected:
        int speed;
        char color[20];
    public:
        virtual void init_speed()
        {
        }
        virtual void init_color()
        {
        }
};

In the above example, both the method `init_speed()` and `init_color()` are overridden in the derived classes called “Dog” and “Tiger”. 
Example in C++ (contd..):

```cpp
class dog:public animal
{
 public:
   void init_speed()
   {
     speed=40;
     cout<<"Speed of a dog is "<<speed<<endl;
   }
   void init_color()
   {
     strcpy(color,"Black");
     cout<<"Color of a dog is "<<color<<endl;
   }
};
```

In the above example, both the method `init_speed()` and `init_color()` are overridden in the derived classes called “Dog” and “Tiger”.
Example in C++ (contd..):

class tiger:public animal
{
 public:
  void init_speed()
  {
   speed=90;
   cout<<"Speed of a Tiger is "<<speed<<endl;
  }
  void init_color()
  {
   strcpy(color,"White");
   cout<<"Color of a Tiger is "<<color<<endl;
  }
};

In the above example, both the method( init_speed() and init_color() ) are overridden in the derived classes called “Dog” and “Tiger”.
Binding

- Process of connecting a method call to a method body.

  - Static
    - Binding is performed before the program runs – Compile time

  - Dynamic
    - Binding is performed at the time of execution – Run time

- A **binding** is an association between an operation and an entity.

- The process of binding can happen either at compile time or at run time.

- Compile time binding is called as **Static Binding**.

- Run time binding is called as **Dynamic Binding**.
Static Binding

- Method calls are bound to specific computer memory locations where methods are placed at compile time
- Leads to run-time efficiency since the compiler can optimize code before executing it
- Function Overloading implements static binding
- Object is going to invoke which overloaded method is decided at the time of compilation by looking at the signature

- Every time a function call is executed, the system will jump to the location where the function code is stored, and will execute the code.

- This is called *static binding or early binding.*

- Static binding is the default behavior for all C++ functions and member functions.

- Overloaded functions are implemented using static binding.

- As binding time gets earlier:
  - efficiency goes up
  - safety goes up
  - flexibility goes down
Dynamic Binding

• The method called depends upon the type of the object and not the type of the expression

• The type cannot always be resolved at compile time
  • Dynamic binding resolves the method to be used at run-time

• Method overriding implements the concept of dynamic binding

• Offers the advantages of flexibility and a high level of problem abstraction

Dynamic Binding:

• Physical behavior is taken on last possible instance - Runtime

Pros and Cons of Dynamic Binding

• Pros
  • Good evolution: Can easily add a new derived class with new method implementations
  • Easy to iterate over similar types of objects

• Cons
  • Performance: Run-time lookup based on type
  • Have to design base classes carefully—Don’t want to break derived classes
  • Harder to understand code—have to study several classes
Dynamic Binding (Contd…)

• OOP language use additional constructs while implementing dynamic binding
  • C++ uses a construct called “Virtual Function”
  • Java uses a construct called “Dynamic method dispatch”

• Have a look at main() function

```c
main(){
animal *p; dog d1;
p=&d1;
p->init_speed();
p->init_color();
}
```
Dynamic Binding Example 1

Example1, showing without the usage of Virtual keyword in C++:

```cpp
#include<iostream.h>
#include<string.h>

class animal
{
    protected:
        int speed;
        char color[20];
    public:
        void init_speed()
        {
            cout<<"Inside Animal Speed()"<<endl;
        }
        void init_color()
        {
            cout<<"Inside Animal Color()"<<endl;
        }
};
```
Dynamic Binding Example 1 (Contd…)

class dog: public animal
{
    public:
        void init_speed()
        {
            speed=40;
            cout<<"Speed of a dog is "<<speed<<endl;
        }
        void init_color()
        {
            strcpy(color,"Black");
            cout<<"Color of a dog is "<<color<<endl;
        }
};

class tiger: public animal
{
Dynamic Binding Example 1 (Contd…)

```cpp
public:
    void init_speed()
    {
        speed=90;
        cout<<"Speed of a Tiger is "<<speed<<endl;
    }
    void init_color()
    {
        strcpy(color,"White");
        cout<<"Color of a Tiger is "<<color<<endl;
    }
};
```
Dynamic Binding Example 1 (Contd…) 

main()
{
    animal *p;
    dog d1;
    tiger t1;
    p=&d1;
    p->init_speed();
    p->init_color();
    p=&t1;
    p->init_speed();
    p->init_color();
}

Output:
Inside Animal Speed()
Inside Animal Color()
Inside Animal Speed()
Inside Animal Color()
Dynamic Binding Example 2

Example 2, showing with the usage of Virtual keyword in C++:
#include<iostream.h>
#include<string.h>
class animal
{
  protected:
    int speed;
    char color[20];
  public:
    virtual void init_speed()
    {
      cout<<"Inside Animal Speed()"<<endl;
    }
    virtual void init_color()
    {
      cout<<"Inside Animal Color()"<<endl;
    }
};
Dynamic Binding Example 2 (contd..)

class dog:public animal
{
public:
    void init_speed()
    {
        speed=40;
        cout<<"Speed of a dog is "<<speed<<endl;
    }
    void init_color()
    {
        strcpy(color,"Black");
        cout<<"Color of a dog is "<<color<<endl;
    }
};
Dynamic Binding Example 2 (contd..)

class tiger:public animal
{
public:
    void init_speed()
    {
        speed=90;
        cout<"Speed of a Tiger is "<<speed<<endl;
    }
    void init_color()
    {
        strcpy(color,"White");
        cout<"Color of a Tiger is "<<color<<endl;
    }
};
Dynamic Binding Example 2 (contd..)

```c
main()
{
    animal *p;
dog d1;
tiger t1;
p=&d1;
p->init_speed();
p->init_color();
p=&t1;
p->init_speed();
p->init_color();
}
```

Output:
- Speed of a dog is 40
- Color of a dog is Black
- Speed of a Tiger is 90
- Color of a Tiger is White

**Note:** Have a look at the output of both the programs.

**Explanation for Non-Virtual methods example.**

In this case the pointer p gets associated with class animal at compile time. So at run time even if the pointer points to an object of dog type or to tiger type, still at the time of invocation of functions it will invoke the methods of animal class.

**Explanation for Virtual methods example.**

At run time it is going to decide based on what object the pointer p points at, the exact version of the function that would be mapped to the function call.

So even if the pointer p is of animal type but at run time it can point to any of it’s derived class object. Method of which class is going to be get invoked depends upon the right hand side object.
Dynamic Binding in Java

- Dynamic Method dispatch – In Java
  - It is the mechanism by which a call to an overridden method is resolved at runtime rather than at compile time.
  
- Just have a look at the statement as r.callme().
  
- Which class callme() is going to invoked will be decided at the time of runtime
  
- Before r.callme() statement, if r is referencing to a memory location of Tiger type then it will invoke method from tiger class
Dynamic Binding Example in Java

class Animal {
    void callme() {
        System.out.println("Inside Animal callme method");
    }
}

class Dog extends Animal {
    // override callme()
    void callme() {
        System.out.println("Inside Dog's callme method");
    }
}

class Tiger extends Animal {
    // override callme()
    void callme() {
        System.out.println("Inside Tiger's callme method");
    }
}
Dynamic Binding Example in Java

class Dispatch {
    public static void main(String args[]) {
        Animal a = new Animal(); // object of type Animal
        Dog b = new Dog(); // object of type Dog
        Tiger c = new Tiger(); // object of type Tiger
        Animal r; // obtain a reference of type Animal

        r = a; // r refers to an Animal object
        r.callme(); // calls Animal's version of callme

        r = b; // r refers to a Dog object
        r.callme(); // calls Dog's version of callme

        r = c; // r refers to a Tiger object
        r.callme(); // calls Tiger's version of callme
    }
}
Advantages of OO Technology

- Shorter development life cycle
- Cost reduction in system development
- Prototyping opportunities that allow early feedback
- Increased scalability
- More suited to the real world
- Hiding the implementation details from the user
- Manage software complexity
Case Study

- Complete the Case Study
Thank You