

# Explain features of JAVA.

Features of java is discussed below:

- **Simple**

Java was designed to be easy for the professional programmer to learn and use effectively. Java is easy to master. If we have Basic concepts of object-oriented programming, learning Java will be even easier. Best of all, if you are an experienced C++ programmer, moving to Java will require very little effort. Because Java inherits the C/C++ syntax and many of the object oriented features of C++, most programmers have little trouble learning Java.

- **Object-Oriented**

Java was not designed to be source-code compatible with any other language. This allowed the Java team the freedom to design with a blank slate. One outcome of this was a clean, usable, pragmatic approach to objects.

- **Robust**

- The multiplatformed environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts you in a few key areas to force you to find your mistakes early in program development. At the same time, Java frees us from having to worry about many of the most common causes of programming errors.
- Because Java is a strictly typed language, it checks your code at compile time. However, it also checks your code at run time. Many hard-to-track-down bugs that often turn up in hard-to reproduce run-time situations are simply impossible to create in Java. Knowing that what you have written will behave in a predictable way under diverse conditions is a key feature of Java.
- To better understand how Java is robust, consider two of the main reasons for program failure: memory management mistakes and mishandled exceptional conditions (that is, run-time errors). Memory management can be a difficult, tedious task in traditional programming environments.

- **Multithreaded**

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows you to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables you to construct smoothly running interactive systems. Java's easy-to-use approach to multithreading allows you to think about the specific behavior of your program, not the multitasking subsystem.

- **Architecture-Neutral**

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow—even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation.

- **Interpreted and High Performance**

Java enables the creation of cross-platform programs by compiling into an intermediate representation called Java bytecode. This code can be executed on any system that implements the Java Virtual Machine. Most previous attempts at cross-platform solutions have done so at the expense of performance. As explained earlier, the Java bytecode was carefully designed so that it would be easy to translate directly into native machine code for very high performance by using a just-in-time compiler. Java run-time systems that provide this feature lose none of the benefits of the platform-independent code.

- **Distributed**

Java is designed for the distributed environment of the Internet because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. Java also supports Remote Method Invocation (RMI). This feature enables a program to invoke methods across a network.

- **Dynamic**

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the Java environment, in which small fragments of bytecode may be dynamically updated on a running system.

## Explain Operators in JAVA.

### *Arithmetic Operator*

Operator	Description	Example
+	Addition - Adds values on either side of the operator	A + B will give 30
-	Subtraction - Subtracts right hand operand from left hand operand	A - B will give -10
*	Multiplication - Multiplies values on either side of the operator	A * B will give 200
/	Division - Divides left hand operand by right hand operand	B / A will give 2
%	Modulus - Divides left hand operand by right hand operand and returns remainder	B % A will give 0
++	Increment - Increases the value of operand by 1	B++ gives 21
--	Decrement - Decreases the value of operand by 1	B-- gives 19

**Arithmetic Operators in JAVA, consider A as 10 & B as 20**

## ***Relational Operators***

<b>Operator</b>	<b>Description</b>	<b>Example</b>
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true.

**Relational Operators in JAVA, consider A as 10 & B as 20**

## ***Bitwise Operators***

<b>Operator</b>	<b>Description</b>	<b>Example</b>
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
	Binary OR Operator copies a bit if it exists in either operand.	(A   B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A ) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>>	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

**Bitwise Operators in JAVA, consider A as 60 & B as 13**

## Logical Operators

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A    B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true.

Logical Operators in JAVA, consider A as true & B as false

## Assignment Operators

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
=	bitwise inclusive OR and assignment operator	C  = 2 is same as C = C   2

Assignment Operators in JAVA

## Explain short circuit operators.

- Java provides two interesting Boolean operators not found in many other computer languages. These are secondary versions of the Boolean AND and OR operators, and are known as shortcircuit logical operators.
- The OR operator results in true when A is true, no matter what B is. Similarly, the AND operator results in false when A is false, no matter what B is. If you use the `||` and `&&` forms, rather than the `|` and `&` forms of these operators, Java will not bother to evaluate the right-hand operand when the outcome of the expression can be determined by the left operand alone.
- This is very useful when the right-hand operand depends on the value of the left one in order to function properly. For example, the following code fragment shows how you can take advantage of short-circuit logical evaluation to be sure that a division operation will be valid before evaluating it:  

```
if (denom != 0 && num / denom > 10)
```
- Since the short-circuit form of AND (`&&`) is used, there is no risk of causing a run-time exception when `denom` is zero. If this line of code were written using the single `&` version of AND, both sides would be evaluated, causing a run-time exception when `denom` is zero. It is standard practice to use the short-circuit forms of AND and OR in cases involving Boolean logic, leaving the single-character versions exclusively for bitwise operations. However, there are exceptions to this rule. For example, consider the following statement:  

```
if(c==1 & e++ < 100) d = 100)
```
- Here, using a single `&` ensures that the increment operation will be applied to `e` whether `c` is equal to 1 or not.

## Explain primitive Data types of JAVA.

Java defines 8 primitive types:

- **byte**
  - Smallest integer type
  - It is a signed 8-bit type (**1 Byte**)
  - Range is -128 to 127
  - Especially useful when working with stream of data from a network or file
  - Example: `byte b = 10;`
- **short**
  - short is signed 16-bit (**2 Byte**) type
  - Range : -32768 to 32767
  - It is probably least used Java type
  - Example: `short vld = 1234;`
- **int**
  - The most commonly used type
  - It is signed 32-bit (**4 Byte**) type
  - Range: -2,147,483,648 to 2,147,483,647
  - Example: `int a = 1234;`
- **long**
  - long is signed 64-bit (**8 Byte**) type

- It is useful when int type is not large enough to hold the desired value
- Example:      long soconds = 1234124231;
- **char**
  - It is 16-bit (**2 Byte**) type
  - Range: 0 to 65,536
  - Example:      char first = 'A';      char second = 65;
- **float**
  - It is 32-bit (**4-Byte**) type
  - It specifies a single-precision value
  - Example:      float price = 1234.45213f
- **double**
  - It uses 64-bit (**8-Byte**)
  - All math functions such as sin(),cos(),sqrt() etc... returns double value
  - Example:      double pi = 3.14141414141414;
- **boolean**
  - The boolean data type has only two possible values: true and false.
  - This data type represents one bit of information, but its "size" isn't something that's precisely defined.

# Arrays in Java

- **One-Dimensional Arrays**

A One-dimensional array is essentially a list of like-typed variables.

Array declaration:

```
type var-name[];
```

Example:

```
int student_marks[];
```

Above example will represent array with no value (null) To link student\_marks with actual, physical array of integers, we must allocate one using new keyword.

Example:

```
int student_marks[] = new int[20];
```

- **Multi-Dimensional Arrays**

A In java, Multidimensional arrays are actually array of arrays.

Example:

```
int runPerOver[][] = new int[50][6];
```

Manually allocate different size:

```
int runPerOver[][] = new int[3][];
```

```
runPerOver[0] = new int[6];
```

```
runPerOver[1] = new int[7];
```

```
runPerOver[2] = new int[6];
```

Initialization :

```
int runPerOver[][] = {  
    {0,4,2,1,0,6},  
    {1,56,4,1,2,4,0},  
    {6,4,1,0,2,2},  
}
```

## Explain String Class in JAVA.

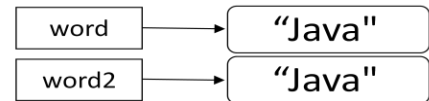
- An object of the String class represents a string of characters.
- The String class belongs to the java.lang package, which does not require an import statement.
- Like other classes, String has constructors and methods.
- Unlike other classes, String has two operators, + and += (used for concatenation).
- Immutability:
  - Once created, a string cannot be changed: none of its methods changes the string.
  - Such objects are called immutable.
  - Immutable objects are convenient because several references can point to the same object safely: there is no danger of changing an object through one reference without the others being aware of the change.
  - Advantages Of Immutability
    - Uses less memory.
  - Disadvantages of Immutability

- Less efficient — you need to create a new string and throw away the old one even for small changes.
- Empty Strings:
  - An empty String has no characters. It's length is 0.
 

```
String word1 = "";
String word2 = new String();
```
  - Not the same as an uninitialized String (null string)
 

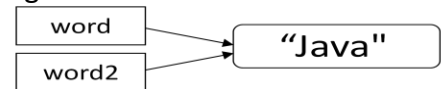
```
private String errorMsg;
```
- Copy Constructors :
  - Copy constructor creates a copy of an existing String. Also rarely used.
  - Not the same as an assignment.
  - Copy Constructor: Each variable points to a different copy of the String.

```
String word = new String("Java");
String word2 = new String(word);
```



- Assignment: Both variables point to the same String.

```
String word = "Java";
String word2 = word;
```



- Other Constructors :
  - Most other constructors take an array as a parameter to create a String.
    - `char[] letters = {'J', 'a', 'v', 'a'};`
    - `String word = new String(letters);`
- Methods of String Class in JAVA :

Method	Description
<code>char charAt(int index)</code>	<code>charAt()</code> function returns the character located at the specified index.
<code>int indexOf(char ch)</code>	Returns the index within this string of the first occurrence of the specified character.
<code>int compareTo(String o)</code>	Compares this String to another Object.
<code>String concat(String str)</code>	Concatenates the specified string to the end of this string.
<code>int length()</code>	The string <code>length()</code> method returns length of the string.
<code>String trim()</code>	The string <code>trim()</code> method eliminates white spaces before and after string.
<code>String replace(char oc, char nc)</code>	The string <code>replace()</code> method replaces all occurrence of first sequence of character with second sequence of character.
<code>toUpperCase()</code>	The java string <code>toUpperCase()</code> method converts this string into uppercase letter.
<code>toLowerCase()</code>	string <code>toLowerCase()</code> method into lowercase letter.

## Explain StringBuffer Class in JAVA.

- The `java.lang.StringBuffer` class is a thread-safe, mutable sequence of characters. Following are the important points about `StringBuffer`:
  - A string buffer is like a `String`, but can be modified.



- It contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.
- They are safe for use by multiple threads.
- Every string buffer has a capacity.

- Class constructors

Sr. No.	Constructor & Description
1	<b>StringBuffer()</b> This constructs a string buffer with no characters in it and an initial capacity of 16 characters.
2	<b>StringBuffer(CharSequence seq)</b> This constructs a string buffer that contains the same characters as the specified CharSequence.
3	<b>StringBuffer(int capacity)</b> This constructs a string buffer with no characters in it and the specified initial capacity.
4	<b>StringBuffer(String str)</b> This constructs a string buffer initialized to the contents of the specified string.

- Here are some of the important methods of StringBuffer class.

- append(arg)
- delete(start index,end index)
- deleteCharAt(index)
- insert(index,arg)
- replace(start index,end index,arg)
- setCharAt(index,newchar)
- reverse()

## Classes in Java.

- A class is a blue print from which individual objects are created.
- A sample of a class is given below:

```
public class Dog{
    String breed;
    int age;
    String color;

    void barking(){
    }

    void hungry(){
    }

    void sleeping(){
    }
}
```

- A class can contain any of the following variable types.
  - **Local variables:** Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
  - **Instance variables:** Instance variables are variables within a class but outside any method. These variables are instantiated when the class is loaded. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
  - **Class variables:** Class variables are variables declared with in a class, outside any method, with the static keyword.
- A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

## Objects in Java.

- Objects have states and behaviors. Example: A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.
- Let us now look deep into what are objects. If we consider the real-world we can find many objects around us, Cars, Dogs, Humans, etc. All these objects have a state and behavior.
- If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging, running
- If you compare the software object with a real world object, they have very similar characteristics.
- Software objects also have a state and behavior. A software object's state is stored in fields and behavior is shown via methods.
- So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

## Constructor in Java.

- Every class has a constructor. If we do not explicitly write a constructor for a class the Java compiler builds a default constructor for that class.
- Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.
- Example of a constructor is given below:

```
public class Puppy{
    public Puppy(){
    }

    public Puppy(String name){
        // This constructor has one parameter, name.
    }
}
```

## Method overloading.

- In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.
- When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.
- Example of Method Overloading by changing the no. of arguments:

```
class Calculation{
    void sum(int a,int b){
        System.out.println(a+b);
    }
    void sum(int a,int b,int c){
        System.out.println(a+b+c);
    }
    public static void main(String args[]){
        Calculation obj=new Calculation();
        obj.sum(10,10,10);
        obj.sum(20,20);
    }
}
```

- Example of Method Overloading by changing data type of argument

```
class Calculation2{
    void sum(int a,int b){
        System.out.println(a+b);
    }
    void sum(double a,double b){
        System.out.println(a+b);
    }
    public static void main(String args[]){
        Calculation2 obj=new Calculation2();
        obj.sum(10.5,10.5);
        obj.sum(20,20);
    }
}
```

## this keyword.

- Java defines the this keyword. this can be used inside any method to refer to the current object. this is always a reference to the object on which the method was invoked.
- To better understand what this refers to, consider the following version of Box( ):

```
// A redundant use of this.
Box(double w, double h, double d) {
    this. width = w;
    this. height = h;
    this. depth = d;
}
```

- The use of this is redundant, but perfectly correct. Inside Box( ), this will always refer to the invoking object.
- While it is redundant in this case, this is useful in other contexts,

```
Box(double width, double height, double depth) {
    this.width = width;
    this.height = height;
    this.depth = depth;
}
```

## static keyword.

- Static means one per class, not one for each object no matter how many instance of a class might exist.
- This means that you can use them without creating an instance of a class.Static methods are implicitly
- final, because overriding is done based on the type of the object, and static methods are attached to a class, not an object.

- A static method in a superclass can be shadowed by another static method in a subclass, as long as the original method was not declared final. However, you can't override a static method with a nonstatic method.
- In other words, you can't change a static method into an instance method in a subclass.

## Access Control in java.

Modifier	Same Class	Same Package	Subclass	Universe
Private	<input checked="" type="checkbox"/>			
Default	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>		
Protected	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
Public	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

## Nested Classes & Inner Classes in Java.

- Java inner class or nested class is a class i.e. declared inside the class or interface.
- Inner class is a part of nested class. Non-static nested classes are known as inner classes.
- We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.
- Additionally, it can access all the members of outer class including private data members and methods.

- Syntax of Inner class:

```
class Java_Outer_class{
    //code
    class Java_Inner_class{
        //code
    }
}
```

- Advantage of java inner classes

There are basically three advantages of inner classes in java. They are as follows:

1. Nested classes represent a special type of relationship that is it can access all the members (data members and methods) of outer class including private.
2. Nested classes are used to develop more readable and maintainable code because it logically group classes and interfaces in one place only.
3. Code Optimization: It requires less code to write.

---

## Abstract Class.

- A Java abstract class is a class which cannot be instantiated, meaning you cannot create new instances of an abstract class.
- The purpose of an abstract class is to function as a base for subclasses.
- Declaring an Abstract Class in Java

```
public abstract class MyAbstractClass {
```

```
}
```

- **Abstract Methods:**

An abstract class can have abstract methods. You declare a method abstract by adding the abstract keyword in front of the method declaration. Here is a Java abstract method example:

```
public abstract class MyAbstractClass {  
    public abstract void abstractMethod();  
}
```

- An abstract method has no implementation. It just has a method signature. Just like methods in a Java interface.
  - If a class has an abstract method, the whole class must be declared abstract. Not all methods in an abstract class have to be abstract methods. An abstract class can have a mixture of abstract and non-abstract methods.
  - Subclasses of an abstract class must implement (override) all abstract methods of its abstract superclass. The non-abstract methods of the superclass are just inherited as they are. They can also be overridden, if needed.
  - **The Purpose of Abstract Classes:**

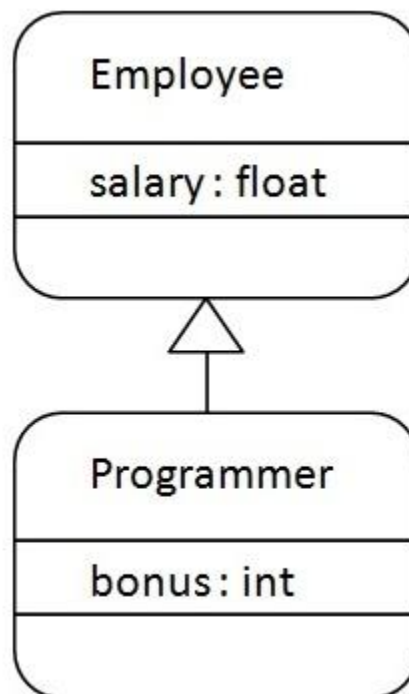
The purpose of abstract classes is to function as base classes which can be extended by subclasses to create a full implementation.
-

# Inheritance in Java

- Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.
- The idea behind inheritance in java is that you can create new classes that are built upon existing classes.
- When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
- Inheritance represents the IS-A relationship, also known as parent-child relationship.
- **Why use inheritance in java**
  - For Method Overriding (so runtime polymorphism can be achieved).
  - For Code Reusability.
- Syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
    //methods and fields
}
```

- The extends keyword indicates that you are making a new class that derives from an existing class.
- In the terminology of Java, a class that is inherited is called a super class. The new class is called a subclass.
- example of inheritance



- As displayed in the above figure, Programmer is the subclass and Employee is the superclass. Relationship between two classes is Programmer IS-A Employee. It means that Programmer is a type of Employee.

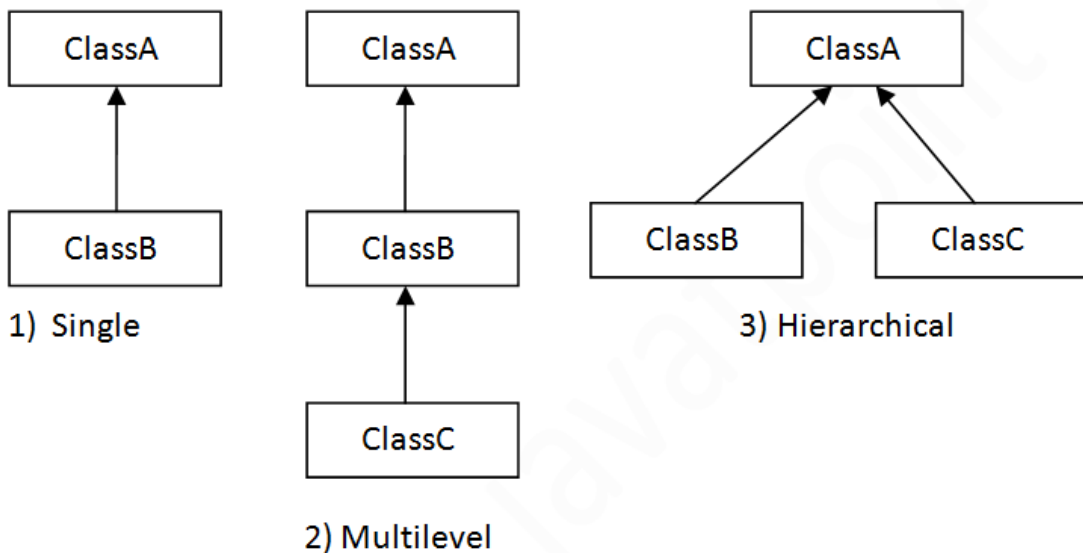
```

class Employee{
    float salary=40000;
}
class Programmer extends Employee{
    int bonus=10000;
    public static void main(String args[]){
        Programmer p=new Programmer();
        System.out.println("Programmer salary is:"+p.salary);
        System.out.println("Bonus of Programmer is:"+p.bonus);
    }
}

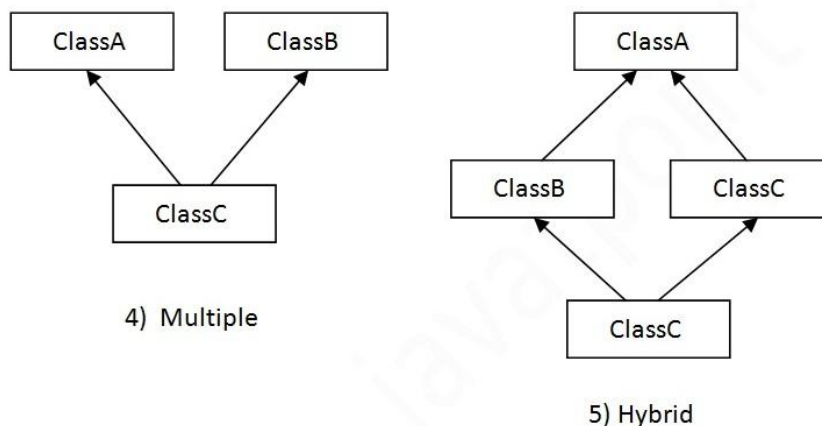
```

- **Types of inheritance in java**

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only.



- Multiple inheritance and Hybrid inheritance is **not** supported in java through class.





## Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B and C are three classes. The C class inherits A and B classes. If A and B classes have same method and you call it from child class object, there will be ambiguity to call method of A or B class.
- Since compile time errors are better than runtime errors, java renders compile time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error now.

```
class A{
    void msg(){System.out.println("Hello");}
}
class B{
    void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

    public static void main(String args[]){
        C obj=new C();
        obj.msg();//Now which msg() method would be invoked?
    }
}
```

## Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to override the method in the superclass.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the superclass will be hidden. Consider the following:

// Method overriding.

```
class A {
    int i, j;
    A(int a, int b) {
        i = a;
        j = b;
    }
    // display i and j
    void show() {
        System.out.println("i and j: " + i + " " + j);
    }
}
```

```

class B extends A {
    int k;
    B(int a, int b, int c) {
        super(a, b);
        k = c;
    }
    // display k – this overrides show() in A
    void show() {
        System.out.println("k: " + k);
    }
}

class Override {
    public static void main(String args[]) {
        B subOb = new B(1, 2, 3);
        subOb.show(); // this calls show() in B
    }
}

```

## Final keyword

- The final keyword in java is used to restrict the user.
- The final keyword can be used in many contexts. Final can be:

### 1. Variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

Example:

```

class Bike9{
    final int speedlimit=90;//final variable
    void run(){
        speedlimit=400;
    }
    public static void main(String args[]){
        Bike9 obj=new Bike9();
        obj.run();
    }
}

```

*Output: Compile Time Error*

### 2. Method

If you make any method as final, you cannot override it.

Example:

```

class Bike{
    final void run(){System.out.println("running");}
}

class Honda extends Bike{
    void run(){

```

```

        System.out.println("running safely with 100kmph");
    }

    public static void main(String args[]){
        Honda honda= new Honda();
        honda.run();
    }
}

```

*Output: Compile Time Error*

### 3. Class

If you make any class as final, you cannot extend it.

Example:

```

    final class Bike{
        // code here
    }

    class Honda1 extends Bike{
        void run(){System.out.println("running safely with 100kmph");}

        public static void main(String args[]){
            Honda1 honda= new Honda();
            honda.run();
        }
    }
}

```

*Output: Compile Time Error*

## Interface in java.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- In practice, this means that you can define interfaces that don't make assumptions about how they are implemented.
- Once it is defined, any number of classes can implement an interface. Also, one class can implement any number of interfaces.
- Syntax of interface:

```

// interface syntax.
Interface myInterface
{
    int a=10;
    public int getBalance();
    public void setBalance(int a);
}

```

- To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation.
- By providing the interface keyword, Java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism.
- Interfaces are designed to support dynamic method resolution at run time.
- Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.
- All the variables defined in the interface are default final and cannot be changed in subclass.

## **instanceof keyword.**

- Sometimes, knowing the type of an object during run time is useful. For example, you might have one thread of execution that generates various types of objects, and another thread that processes these objects.
- In this situation, it might be useful for the processing thread to know the type of each object when it receives it. Another situation in which knowledge of an object’s type at run time is important involves casting.
- In Java, an invalid cast causes a run-time error. Many invalid casts can be caught at compile time. However, casts involving class hierarchies can produce invalid casts that can be detected only at run time.
- For example, a superclass called A can produce two subclasses, called B and C. Thus, casting a B object into type A or casting a C object into type A is legal, but casting a B object into type C (or vice versa) isn’t legal. Because an object of type A can refer to objects of either B or C, how can you know, at run time, what type of object is actually being referred to before attempting the cast to type C? It could be an object of type A, B, or C. If it is an object of type B, a run-time exception will be thrown. Java provides the run-time operator instanceof to answer this question.
- The instanceof operator has this general form:  
object instanceof type
- The following program demonstrates instanceof:

```
class A
{
    public static void main(String[] ar)
    {
        String a = "hello";
        Double d = new Double(10.0);
        If(a instanceof String)
        {
            System.out.println("A is a instance of String");
        }

        If(d instanceof Object)
        {
            System.out.println("D is a instance of Object");
        }
    }
}
```

```
}  
}
```

## Dynamic Method Dispatch.

- Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.
- Let's begin by restating an important principle: a superclass reference variable can refer to a subclass object. Java uses this fact to resolve calls to overridden methods at run time. Here is how. When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. When different types of objects are referred to, different versions of an overridden method will be called.
- In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed. Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.
- Here is an example that illustrates dynamic method dispatch:

```
// Dynamic Method Dispatch  
class A {  
    void callme() {  
        System.out.println("Inside A's callme method");  
    }  
}  
  
class B extends A {  
    // override callme()  
    void callme() {  
        System.out.println("Inside B's callme method");  
    }  
}  
  
class C extends A {  
    // override callme()  
    void callme() {  
        System.out.println("Inside C's callme method");  
    }  
}  
  
class Dispatch {  
    public static void main(String args[]) {  
        A a = new A(); // object of type A
```

```

    B b = new B(); // object of type B
    C c = new C(); // object of type C

    A r; // obtain a reference of type A
    r = a; // r refers to an A object
    r.callme(); // calls A's version of callme
    r = b; // r refers to a B object
    r.callme(); // calls B's version of callme
    r = c; // r refers to a C object
    r.callme(); // calls C's version of callme
}
}

```

Output:

Inside A's callme method

Inside B's callme method

Inside C's callme method

- This program creates one superclass called A and two subclasses of it, called B and C. Subclasses B and C override callme( ) declared in A. Inside the main( ) method, objects of type A, B, and C are declared. Also, a reference of type A, called r, is declared. The program then in turn assigns a reference to each type of object to r and uses that reference to invoke callme( ). As the output shows, the version of callme( ) executed is determined by the type of object being referred to at the time of the call. Had it been determined by the type of the reference variable, r, you would see three calls to A's callme( ) method.

## Comparison between Abstract Class and interface.

Abstract Class	Interface
A Java abstract class can have instance methods that implement a default behavior.	Java interface are implicitly abstract and cannot have implementations
An abstract class may contain non-final variables.	Variables declared in a Java interface is by default final
A Java abstract class can have the usual flavors of class members like private, protected, etc..	Members of a Java interface are public by default
A Java abstract class should be extended using keyword "extends".	Java interface should be implemented using keyword "implements".
A Java class can extend only one abstract class.	A Java class can implement multiple interfaces

## Use of Package

- Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.
- A Package can be defined as a grouping of related types(classes, interfaces, enumerations and annotations ) providing access protection and name space management.
- Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, annotations are related.
- Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

## Creating a package:

- When creating a package, you should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.
- The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.
- If a package statement is not used then the class, interfaces, enumerations, and annotation types will be put into an unnamed package.
- Example:

```
package animals;

interface Animal {
    public void eat();
}
```

## The import Keyword:

- If a class wants to use another class in the same package, the package name does not need to be used. Classes in the same package find each other without any special syntax.
- Example:

```
import animals.Animal;
// OR import animals.*; to import all the classes in the animals package

public class Dog implements Animal
{
    public void ear()
    {
        // Code Here
    }
}
```

## Set CLASSPATH System Variable:

- To display the current CLASSPATH variable, use the following commands in Windows and UNIX (Bourne shell):
  - In Windows -> C:\> set CLASSPATH
  - In UNIX -> % echo \$CLASSPATH
- To delete the current contents of the CLASSPATH variable, use :
  - In Windows -> C:\> set CLASSPATH=
  - In UNIX -> % unset CLASSPATH; export CLASSPATH
- To set the CLASSPATH variable:
  - In Windows -> set CLASSPATH=C:\users\jack\java\classes
  - In UNIX -> % CLASSPATH=/home/jack/java/classes; export CLASSPATH

## Static Import:

- The static import feature of Java 5 facilitate the java programmer to access any static member of a class directly. There is no need to qualify it by the class name.
- Advantage of static import:
  - Less coding is required if you have access any static member of a class oftenly.
- Disadvantage of static import:
  - If you overuse the static import feature, it makes the program unreadable and unmaintainable.

- Example of static import:

```
import static java.lang.System.*;
class StaticImportExample{
    public static void main(String args[]){

        out.println("Hello");//Now no need of System.out
        out.println("Java");

    }
}
```

## What is the difference between import and static import?

The import allows the java programmer to access classes of a package without package qualification whereas the static import feature allows to access the static members of a class without the class qualification. The import provides accessibility to classes and interface whereas static import provides accessibility to static members of the class.



## Difference between error and exception

- Errors and exceptions both inherit from Throwable, but they differ in these ways:

Exceptions	Errors
<ul style="list-style-type: none"><li>Any departure from the expected behavior of the system or program, which stops the working of the system is an error.</li></ul>	<ul style="list-style-type: none"><li>Any error or problem which one can handle and continue to work normally.</li></ul>
<ul style="list-style-type: none"><li>Can be checked or unchecked</li></ul>	<ul style="list-style-type: none"><li>Are always unchecked</li></ul>
<ul style="list-style-type: none"><li>Indicate an error caused by the programmer</li></ul>	<ul style="list-style-type: none"><li>Usually indicate a system error or a problem with a low-level resource</li></ul>
<ul style="list-style-type: none"><li>Should be handled at the application level</li></ul>	<ul style="list-style-type: none"><li>Should be handled at the system level, if possible</li></ul>

## Checked and Unchecked Exceptions.

Exceptions in Java are classified on the basis of the exception handled by the java compiler. Java consists of the following type of built in exceptions:

- Checked Exception:** These exception are the object of the Exception class or any of its subclasses except Runtime Exception class. These condition arises due to invalid input, problem with your network connectivity and problem in database.java.io.IOException is a checked exception. This exception is thrown when there is an error in input-output operation. In this case operation is normally terminated.

- List of Checked Exceptions

Exception	Reason for Exception
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program
Instantiation Exception	This Exception occurs when you create an object of an abstract class and interface
Illegal Access Exception	This Exception occurs when you create an object of an abstract class and interface
Not Such Method Exception	This Exception occurs when the method you call does not exist in class

- **Unchecked Exception:** These Exception arises during run-time ,that occur due to invalid argument passed to method. The java Compiler does not check the program error during compilation. For Example when you divide a number by zero, run-time exception is raised.
  - List of Unchecked Exceptions

Exception	Reason for Exception
Arithmetic Exception	These Exception occurs, when you divide a number by zero causes an Arithmetic Exception
Array Store Exception	These Exception occurs, when you assign an array which is not compatible with the data type of that array
Array Index Out Of Bounds Exception	These Exception occurs, when you assign an array which is not compatible with the data type of that array
Null Pointer Exception	These Exception occurs, when you try to implement an application without referencing the object and allocating to a memory
Number Format Exception	These Exception occurs, when you try to convert a string variable in an incorrect format to integer (numeric format) that is not compatible with each other

## Explain use of throw.

- It is possible for your program to throw an exception explicitly, using the throw statement. The general form of throw is shown here:  
    throw ThrowableInstance;
- Here, ThrowableInstance must be an object of type Throwable or a subclass of Throwable. Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:
  - using a parameter in a catch clause, or creating one with the new operator. The flow of execution stops immediately after the throw statement; any subsequent statements are not executed. The nearest enclosing try block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not, then the next enclosing try statement is inspected, and so on. If no matching catch is found, then the default exception handler halts the program and prints the stack trace.
- Here is a sample program that creates and throws an exception. The handler that catches the exception rethrows it to the outer handler.

- Example:

```
// Demonstrate throw.
```

```
class ThrowDemo {
    static void demoproc() {
        try
        {
            throw new NullPointerException("demo");
        } catch(NullPointerException e) {
            System.out.println("Caught inside demoproc.");
            throw e; // rethrow the exception
        }
    }
    public static void main(String args[]) {
        try {
            demoproc();
        } catch(NullPointerException e) {
            System.out.println("Recaught: " + e);
        }
    }
}
```

- This program gets two chances to deal with the same error. First, main( ) sets up an exception context and then calls demoproc( ). The demoproc( ) method then sets up another exceptionhandling context and immediately throws a new instance of NullPointerException, which is caught on the next line. The exception is then rethrown. Here is the resulting output:

```
Caught inside demoproc.
```

```
Recaught: java.lang.NullPointerException: demo
```

The program also illustrates how to create one of Java's standard exception objects.

- Pay close attention to this line:

```
throw new NullPointerException("demo");
```

- Here, new is used to construct an instance of NullPointerException. Many of Java's builtin run-time exceptions have at least two constructors: one with no parameter and one that takes a string parameter. When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to print( ) or println( ). It can also be obtained by a call to getMessage( ), which is defined by Throwable.

## Explain use of throws.

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a throws clause in the method's declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses. All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result. This is the general form of a method declaration that includes a throws clause:

```
type method-name(parameter-list) throws exception-list
```

```
{  
    // body of method  
}
```

- Here, exception-list is a comma-separated list of the exceptions that a method can throw. Following is an example of an incorrect program that tries to throw an exception that it does not catch. Because the program does not specify a throws clause to declare this fact, the program will not compile.

```
// This program contains an error and will not compile.  
class ThrowsDemo {  
    static void throwOne() {  
        System.out.println("Inside throwOne.");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        throwOne();  
    }  
}
```

- To make this example compile, you need to make two changes. First, you need to declare that throwOne( ) throws IllegalAccessException. Second, main( ) must define a try/catch statement that catches this exception.

The **corrected** example is shown here:

```
// This is now correct.  
class ThrowsDemo {  
    static void throwOne() throws IllegalAccessException {  
        System.out.println("Inside throwOne.");  
        throw new IllegalAccessException("demo");  
    }  
    public static void main(String args[]) {  
        try {  
            throwOne();  
        } catch (IllegalAccessException e) {  
            System.out.println("Caught " + e);  
        }  
    }  
}
```

## Explain use of finally.

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The finally keyword is designed to address this contingency.

- Finally creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block. The finally block will execute whether or not an exception is thrown. If an exception is thrown, the finally block will execute even if no catch statement matches the exception.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning. The finally clause is optional. However, each try statement requires at least one catch or a finally clause.

Here is an example program that shows three methods that exit in various ways, none without executing their finally clauses:

```
// Demonstrate finally.
class FinallyDemo {
    // Through an exception out of the method.
    static void procA() {
        try {
            System.out.println("inside procA");
            throw new RuntimeException("demo");
        } finally {
            System.out.println("procA's finally");
        }
    }
    // Return from within a try block.
    static void procB() {
        try {
            System.out.println("inside procB");
            return;
        } finally {
            System.out.println("procB's finally");
        }
    }
    // Execute a try block normally.
    static void procC() {
        try {
            System.out.println("inside procC");
        } finally {
            System.out.println("procC's finally");
        }
    }
    public static void main(String args[]) {
        try {
            procA();
        } catch (Exception e) {
```

```
        System.out.println("Exception caught");
    }
    procB();
    procC();
}
}
```

- In this example, `procA( )` prematurely breaks out of the try by throwing an exception. The finally clause is executed on the way out. `procB( )`'s try statement is exited via a return statement. The finally clause is executed before `procB( )` returns. In `procC( )`, the try statement executes normally, without error. However, the finally block is still executed.

**Output:**

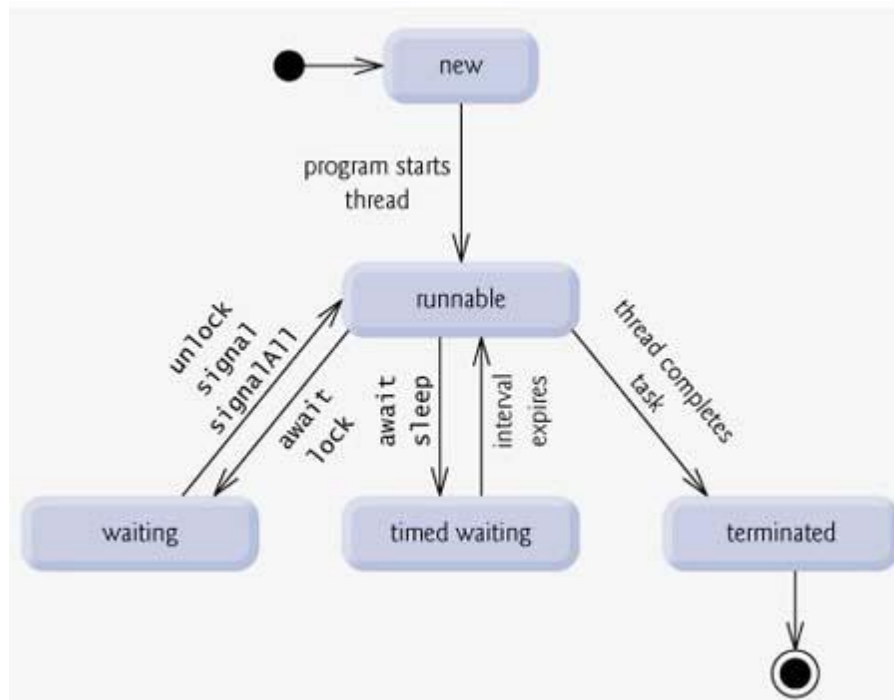
```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

## What is multithreading? Why it is required?

- Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. A multithreading is a specialized form of multitasking. Multitasking threads require less overhead than multitasking processes.
- I need to define another term related to threads: process: A process consists of the memory space allocated by the operating system that can contain one or more threads. A thread cannot exist on its own; it must be a part of a process. A process remains running until all of the non-daemon threads are done executing.
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

## Explain the life cycle of a thread

- A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



- Above mentioned stages are explained here:
  - **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
  - **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
  - **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.

- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

## Create Thread by Implementing Runnable Interface:

- The easiest way to create a thread is to create a class that implements the Runnable interface. To implement Runnable, a class need only implement a single method called run(), which is declared like this:

```
public void run( )
```

- You will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.
- After creating a class that implements Runnable will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

- Here threadOb is an instance of a class that implements the Runnable interface and the name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its start( ) method, which is declared within Thread. The start( ) method is shown here:

```
void start( );
```

- Example:

Here is an example that creates a new thread and starts it running:

```
// Create a new thread.
```

```
class NewThread implements Runnable {
```

```
    Thread t;
```

```
    NewThread() {
```

```
        // Create a new, second thread
```

```
        t = new Thread(this, "Demo Thread");
```

```
        System.out.println("Child thread: " + t);
```

```
        t.start(); // Start the thread
```

```
    }
```

```
// This is the entry point for the second thread.
```

```
public void run() {
```

```
    try {
```

```
        for(int i = 5; i > 0; i--) {
```

```
            System.out.println("Child Thread: " + i);
```

```
            // Let the thread sleep for a while.
```

```
            Thread.sleep(500);
```

```
        }
```

```
    } catch (InterruptedException e) {
```

```
        System.out.println("Child interrupted.");
```

```
    }
```



```

        System.out.println("Exiting child thread.");
    }
}

class ThreadDemo {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Output:

```

Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread: 1
Main thread exiting.

```

## Create Thread by Extending Thread Class:

- The second way to create a thread is to create a new class that extends Thread, and then to create an instance of that class. The extending class must override the run( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.
- Example:
 

```

// Create a second thread by extending Thread
class NewThread extends Thread {
    NewThread() {
        // Create a new, second thread

```

```

        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start(); // Start the thread
    }

    // This is the entry point for the second thread.
    public void run() {
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Child Thread: " + i);
                // Let the thread sleep for a while.
                Thread.sleep(500);
            }
        } catch (InterruptedException e) {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}

class ExtendThread {
    public static void main(String args[]) {
        new NewThread(); // create a new thread
        try {
            for(int i = 5; i > 0; i--) {
                System.out.println("Main Thread: " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Output:

```

Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1

```

Exiting child thread.

Main Thread: 2

Main Thread: 1

Main thread exiting.

## Priority of a Thread (Thread Priority):

- Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread scheduler schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.
- 3 constants defined in Thread class:
  - `public static int MIN_PRIORITY`
  - `public static int NORM_PRIORITY`
  - `public static int MAX_PRIORITY`
- Default priority of a thread is 5 (`NORM_PRIORITY`). The value of `MIN_PRIORITY` is 1 and the value of `MAX_PRIORITY` is 10.
- Example of priority of a Thread:

```
class TestMultiPriority1 extends Thread{
    public void run(){
        System.out.println("running thread name is:"+Thread.currentThread().getName());
        System.out.println("running priority is:"+Thread.currentThread().getPriority());

    }
    public static void main(String args[]){
        TestMultiPriority1 m1=new TestMultiPriority1();
        TestMultiPriority1 m2=new TestMultiPriority1();
        m1.setPriority(Thread.MIN_PRIORITY);
        m2.setPriority(Thread.MAX_PRIORITY);
        m1.start();
        m2.start();
    }
}
```

## Why synchronization is required in multithreaded programming and how can we implement it in program?

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this synchronization is achieved is called thread synchronization. The `synchronized` keyword in Java creates a block of code referred to as a critical section. Every Java object with a critical section of code gets a lock associated with the object. To enter a critical section, a thread needs to obtain the corresponding object's lock.

- Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.
- Here is an example, using a synchronized block within the run( ) method:

This would produce following result:

```
// This program uses a synchronized block.
class Callme {
    void call(String msg) {
        System.out.print "[" + msg);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
        System.out.println("]");
    }
}

// File Name : Caller.java
class Caller implements Runnable {
    String msg;
    Callme target;
    Thread t;
    public Caller(Callme targ, String s) {
        target = targ;
        msg = s;
        t = new Thread(this);
        t.start();
    }

    // synchronize calls to call()
    public void run() {
        synchronized(target) { // synchronized block
            target.call(msg);
        }
    }
}

// File Name : Synch.java
class Synch {
    public static void main(String args[]) {
        Callme target = new Callme();
        Caller ob1 = new Caller(target, "Hello");
        Caller ob2 = new Caller(target, "Synchronized");
        Caller ob3 = new Caller(target, "World");
    }
}
```

```

        // wait for threads to end
        try {
            ob1.t.join();
            ob2.t.join();
            ob3.t.join();
        } catch (InterruptedException e) {
            System.out.println("Interrupted");
        }
    }
}

```

Output:

```

[Hello]
[World]
[Synchronized]

```

## Explain interprocess communication mechanism

- To avoid polling, Java includes an elegant interprocess communication mechanism via the `wait()`, `notify()`, and `notifyAll()` methods. These methods are implemented as final methods in `Object`, so all classes have them. All three methods can be called only from within a synchronized context. Although conceptually advanced from a computer science perspective, the rules for using these methods are actually quite simple:
  - **`wait()`** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls `notify()`.
  - **`notify()`** wakes up the first thread that called `wait()` on the same object.
  - **`notifyAll()`** wakes up all the threads that called `wait()` on the same object. The highest priority thread will run first.

These methods are declared within `Object`, as shown here:

```

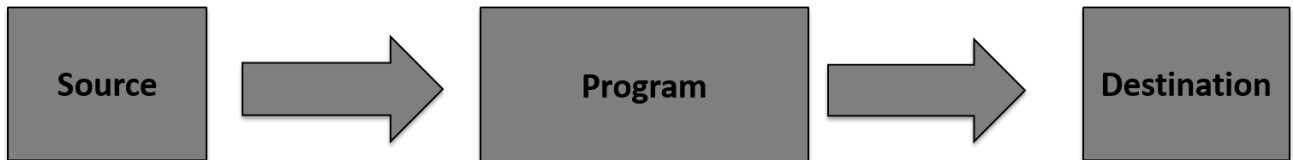
final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )

```

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, Object, localized characters, etc.

## Stream

- A stream can be defined as a sequence of data. there are two kinds of Streams
  - **InPutStream** : The InputStream is used to read data from a source.
  - **OutPutStream** : The OutputStream is used for writing data to a destination.



- Java provides strong but flexible support for I/O related to Files and networks but this tutorial covers very basic functionality related to streams and I/O.

## Byte Streams

- Java byte streams are used to perform input and output of 8-bit bytes.
- Though there are many classes related to byte streams but the most frequently used classes are `FileInputStream` and `FileOutputStream`.
- Following is an example which makes use of these two classes to copy an input file into an output file:

```
import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

```

    }
}
}

```

## Character Streams

- Java Byte streams are used to perform input and output of 8-bit bytes, where as Java Character streams are used to perform input and output for 16-bit unicode.
- Though there are many classes related to character streams but the most frequently used classes are `FileReader` and `FileWriter`.
- Though internally `FileReader` uses `FileInputStream` and `FileWriter` uses `FileOutputStream` but here major difference is that `FileReader` reads two bytes at a time and `FileWriter` writes two bytes at a time.
- We can re-write above example which makes use of these two classes to copy an input file (having unicode characters) into an output file:
- Example:

```

import java.io.*;
public class CopyFile {
    public static void main(String args[]) throws IOException {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}

```

## Standard Streams

- All the programming languages provide support for standard I/O where user's program can take input from a keyboard and then produce output on the computer screen.

- If you are aware if C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similar way Java provides following three standard streams.
  - **Standard Input:** This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.
  - **Standard Output:** This is used to output the data produced by the user's program and usually a computer screen is used to standard output stream and represented as **System.out**.
  - **Standard Error:** This is used to output the error data produced by the user's program and usually a computer screen is used to standard error stream and represented as **System.err**.

## Reader

- The Java Reader is the base class of all Reader's in the Java IO API. Subclasses include a BufferedReader, PushbackReader, InputStreamReader, StringReader and several others.
- Here is a simple Java IO Reader example:

```
Reader reader = new FileReader("c:\\data\\myfile.txt");

int data = reader.read();

while(data != -1){

    char dataChar = (char) data;

    data = reader.read();

}
```

- Combining Readers With InputStreams

```
Reader reader = new InputStreamReader(inputStream);
```

## Writer

- The Java Writer class is the base class of all Writers in the Java IO API. Subclasses include BufferedWriter and PrintWriter among others.
- Here is a simple Java IO Writer example:

```
Writer writer = new FileWriter("c:\\data\\file-output.txt");

writer.write("Hello World Writer");
```



```
writer.close();
```

- Combining Readers With InputStreams

```
Writer writer = new OutputStreamWriter(outputStream);
```

## File Class

- Java File class represents the files and directory pathnames in an abstract manner. This class is used for creation of files and directories, file searching, file deletion etc.
- The File object represents the actual file/directory on the disk. Below given is the list of constructors to create a File object.

SR.NO	Methods with Description
1	<b>File(File parent, String child)</b> This constructor creates a new File instance from a parent abstract pathname and a child pathname string.
2	<b>File(String pathname)</b> This constructor creates a new File instance by converting the given pathname string into an abstract pathname.
3	<b>File(String parent, String child)</b> This constructor creates a new File instance from a parent pathname string and a child pathname string.
4	<b>File(URI uri)</b> This constructor creates a new File instance by converting the given file: URI into an abstract pathname.

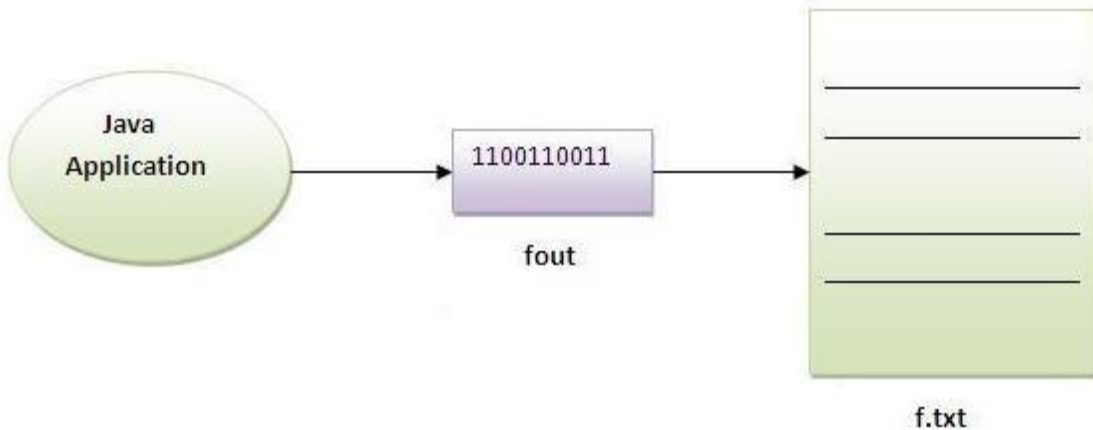
- Once you have File object in hand then there is a list of helper methods which can be used to manipulate the files.

SR.NO	Methods with Description
1	<b>public boolean isAbsolute()</b> Tests whether this abstract pathname is absolute. Returns true if this abstract pathname is absolute, false otherwise
2	<b>public String getAbsolutePath()</b> Returns the absolute pathname string of this abstract pathname.

3	<b>public boolean canRead()</b> Tests whether the application can read the file denoted by this abstract pathname. Returns true if and only if the file specified by this abstract pathname exists and can be read by the application; false otherwise.
4	<b>public boolean canWrite()</b> Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if and only if the file system actually contains a file denoted by this abstract pathname and the application is allowed to write to the file; false otherwise.
5	<b>public boolean exists()</b> Tests whether the file or directory denoted by this abstract pathname exists. Returns true if and only if the file or directory denoted by this abstract pathname exists; false otherwise
6	<b>public boolean isDirectory()</b> Tests whether the file denoted by this abstract pathname is a directory. Returns true if and only if the file denoted by this abstract pathname exists and is a directory; false otherwise.
7	<b>public boolean isFile()</b> Tests whether the file denoted by this abstract pathname is a normal file. A file is normal if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file. Returns true if and only if the file denoted by this abstract pathname exists and is a normal file; false otherwise .
8	<b>public long lastModified()</b> Returns the time that the file denoted by this abstract pathname was last modified. Returns a long value representing the time the file was last modified, measured in milliseconds since the epoch (00:00:00 GMT, January 1, 1970).
9	<b>public long length()</b> Returns the length of the file denoted by this abstract pathname. The return value is unspecified if this pathname denotes a directory.
10	<b>public boolean delete()</b> Deletes the file or directory denoted by this abstract pathname. If this pathname denotes a directory, then the directory must be empty in order to be deleted.
11	<b>public String[] list()</b> Returns an array of strings naming the files and directories in the directory denoted by this abstract pathname.

# FileOutputStream

- Java FileOutputStream is an output stream for writing data to a file.
- If you have to write primitive values then use FileOutputStream. Instead, for character-oriented data, prefer FileWriter. But you can write byte-oriented as well as character-oriented data.

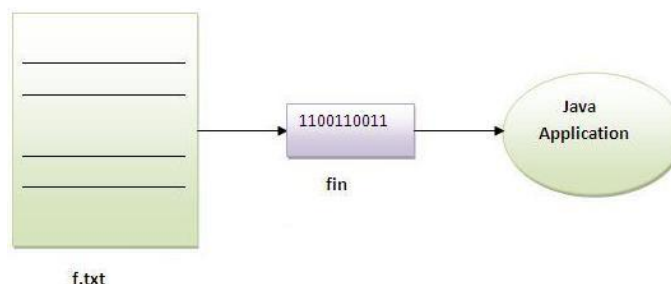


- Example :

```
import java.io.*;
class Test{
    public static void main(String args[]){
        try{
            FileOutputStream fout=new FileOutputStream("abc.txt");
            String s="Sachin Tendulkar is my favourite player";
            byte b[]=s.getBytes();//converting string into byte array
            fout.write(b);
            fout.close();
            System.out.println("success...");
        }catch(Exception e)
        {
            system.out.println(e);
        }
    }
}
```

# FileInputStream

- Java FileInputStream class obtains input bytes from a file. It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.
- It should be used to read byte-oriented data for example to read image, audio, video etc.



- Example :

```
import java.io.*;
class SimpleRead{
    public static void main(String args[]){
        try{
            FileInputStream fin=new FileInputStream("abc.txt");
            int i=0;
            while((i=fin.read())!=-1){
                System.out.println((char)i);
            }
            fin.close();
        }
        catch(Exception e){
            system.out.println(e);
        }
    }
}
```

## InputStreamReader

- The Java.io.InputStreamReader class is a bridge from byte streams to character streams.It reads bytes and decodes them into characters using a specified charset.
- Constructors:

S.N.	Constructor & Description
1	<b>InputStreamReader(InputStream in)</b> This creates an InputStreamReader that uses the default charset.
2	<b>InputStreamReader(InputStream in, Charset cs)</b> This creates an InputStreamReader that uses the given charset.
3	<b>InputStreamReader(InputStream in, CharsetDecoder dec)</b> This creates an InputStreamReader that uses the given charset decoder.
4	<b>InputStreamReader(InputStream in, String charsetName)</b> This creates an InputStreamReader that uses the named charset.

- Methods:

S.N.	Method & Description
1	<b>void close()</b> This method closes the stream and releases any system resources associated with it.

2	<b>String getEncoding()</b> This method returns the name of the character encoding being used by this stream.
3	<b>int read()</b> This method reads a single character.
4	<b>int read(char[] cbuf, int offset, int length)</b> This method reads characters into a portion of an array.
5	<b>boolean ready()</b> This method tells whether this stream is ready to be read.

## OutputStreamWriter

- The `Java.io.OutputStreamWriter` class is a bridge from character streams to byte streams. Characters written to it are encoded into bytes using a specified charset.
- Constructors:

S.N.	Constructor & Description
1	<b>OutputStreamWriter(OutputStream out)</b> This creates an <code>OutputStreamWriter</code> that uses the default character encoding.
2	<b>OutputStreamWriter(OutputStream out, Charset cs)</b> This creates an <code>OutputStreamWriter</code> that uses the given charset.
3	<b>OutputStreamWriter(OutputStream out, CharsetEncoder enc)</b> This creates an <code>OutputStreamWriter</code> that uses the given charset encoder.
4	<b>OutputStreamWriter(OutputStream out, String charsetName)</b> This creates an <code>OutputStreamWriter</code> that uses the named charset.

- Methods:

S.N.	Method & Description
1	<b>void close()</b> This method closes the stream, flushing it first.
2	<b>void flush()</b>

	This method flushes the stream.
3	<b>String getEncoding()</b> This method returns the name of the character encoding being used by this stream.
4	<b>void write(char[] cbuf, int off, int len)</b> This method writes a portion of an array of characters.
5	<b>void write(int c)</b> This method writes a single character.
6	<b>void write(String str, int off, int len)</b> This method writes a portion of a string.

## FileReader

- This class inherits from the InputStreamReader class. FileReader is used for reading streams of characters.
- This class has several constructors to create required objects. Below given are the list of constructors provided by the FileReader class.
- Contractors:

SR.NO	Constructors and Description
1	<b>FileReader(File file)</b> This constructor creates a new FileReader, given the File to read from.
2	<b>FileReader(FileDescriptor fd)</b> This constructor creates a new FileReader, given the FileDescriptor to read from.
3	<b>FileReader(String fileName)</b> This constructor creates a new FileReader, given the name of the file to read from.

- Methods:

SR.NO	Methods with Description
-------	--------------------------

1	<b>public int read() throws IOException</b>  Reads a single character. Returns an int, which represents the character read.
2	<b>public int read(char [] c, int offset, int len)</b>  Reads characters into an array. Returns the number of characters read.

- Example:

```
import java.io.*;

public class FileRead{

    public static void main(String args[])throws IOException{

        File file = new File("Hello1.txt");
        // creates the file
        file.createNewFile();
        // creates a FileWriter Object
        FileWriter writer = new FileWriter(file);
        // Writes the content to the file
        writer.write("This\n is\n an\n example\n");
        writer.flush();
        writer.close();

        //Creates a FileReader Object
        FileReader fr = new FileReader(file);
        char [] a = new char[50];
        fr.read(a); // reads the content to the array
        for(char c : a)
            System.out.print(c); //prints the characters one by one
        fr.close();
    }
}
```

# FileWriter

- This class inherits from the `OutputStreamWriter` class. The class is used for writing streams of characters.
- This class has several constructors to create required objects. Below given is the list of them
- Contractors:

SR.NO	Constructors and Description
1	<b>FileWriter(File file)</b> This constructor creates a <code>FileWriter</code> object given a <code>File</code> object.
2	<b>FileWriter(File file, boolean append)</b> This constructor creates a <code>FileWriter</code> object given a <code>File</code> object. with a boolean indicating whether or not to append the data written.
3	<b>FileWriter(FileDescriptor fd)</b> This constructor creates a <code>FileWriter</code> object associated with the given file descriptor.
4	<b>FileWriter(String fileName)</b> This constructor creates a <code>FileWriter</code> object, given a file name.
5	<b>FileWriter(String fileName, boolean append)</b> This constructor creates a <code>FileWriter</code> object given a file name with a boolean indicating whether or not to append the data written.

- Methods:

SN	Methods with Description
1	<b>public void write(int c) throws IOException</b> Writes a single character.
2	<b>public void write(char [] c, int offset, int len)</b> Writes a portion of an array of characters starting from offset and with a length of len.
3	<b>public void write(String s, int offset, int len)</b> Write a portion of a <code>String</code> starting from offset and with a length of len.



# BufferedReader

- The `Java.io.BufferedReader` class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines. Following are the important points about `BufferedReader`:
  - The buffer size may be specified, or the default size may be used.
  - Each read request made of a `Reader` causes a corresponding read request to be made of the underlying character or byte stream.
- Constructors:

S.N.	Constructor & Description
1	<b><code>BufferedReader(Reader in)</code></b> This creates a buffering character-input stream that uses a default-sized input buffer.
2	<b><code>BufferedReader(Reader in, int sz)</code></b> This creates a buffering character-input stream that uses an input buffer of the specified size.

- Methods:

S.N.	Method & Description
1	<b><code>void close()</code></b> This method closes the stream and releases any system resources associated with it.
2	<b><code>int read()</code></b> This method reads a single character.
3	<b><code>int read(char[] cbuf, int off, int len)</code></b> This method reads characters into a portion of an array.
4	<b><code>String readLine()</code></b> This method reads a line of text.
5	<b><code>void reset()</code></b> This method resets the stream.
6	<b><code>long skip(long n)</code></b> This method skips characters.



## List Interface.

- The List interface extends Collection and declares the behavior of a collection that stores a sequence of elements.
  - Elements can be inserted or accessed by their position in the list, using a zero-based index.
  - A list may contain duplicate elements.
  - In addition to the methods defined by Collection, List defines some of its own, which are summarized in the following below Table.
  - Several of the list methods will throw an UnsupportedOperationException if the collection cannot be modified, and a ClassCastException is generated when one object is incompatible with another.
- Methods :

SN	Methods with Description
1	<b>void add(int index, Object obj)</b> Inserts obj into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	<b>boolean addAll(int index, Collection c)</b> Inserts all elements of c into the invoking list at the index passed in index. Any pre-existing elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten. Returns true if the invoking list changes and returns false otherwise.
3	<b>Object get(int index)</b> Returns the object stored at the specified index within the invoking collection.
4	<b>int indexOf(Object obj)</b> Returns the index of the first instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
5	<b>int lastIndexOf(Object obj)</b> Returns the index of the last instance of obj in the invoking list. If obj is not an element of the list, .1 is returned.
6	<b>ListIterator listIterator( )</b> Returns an iterator to the start of the invoking list.
7	<b>ListIterator listIterator(int index)</b> Returns an iterator to the invoking list that begins at the specified index.
8	<b>Object remove(int index)</b>

	Removes the element at position index from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one
9	<b>Object set(int index, Object obj)</b> Assigns obj to the location specified by index within the invoking list.
10	<b>List subList(int start, int end)</b> Returns a list that includes elements from start to end-1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

- A sample of a List is given below:

```
import java.util.*;

public class CollectionsDemo {

    public static void main(String[] args) {

        List a1 = new ArrayList();
        a1.add("Zara");
        a1.add("Mahnaz");
        a1.add("Ayan");
        System.out.println(" ArrayList Elements");
        System.out.print("\t" + a1);

        List l1 = new LinkedList();
        l1.add("Zara");
        l1.add("Mahnaz");
        l1.add("Ayan");
        System.out.println();
        System.out.println(" LinkedList Elements");
        System.out.print("\t" + l1);

    }
}
```

## ArrayList Class.

- The java.util.ArrayList class provides resizable-array and implements the List interface. Following are the important points about ArrayList:

- It implements all optional list operations and it also permits all elements, includes null.
- It provides methods to manipulate the size of the array that is used internally to store the list.
- The constant factor is low compared to that for the LinkedList implementation.

- Constructors:

S.N.	Constructor & Description
1	<b>ArrayList()</b> This constructor is used to create an empty list with an initial capacity sufficient to hold 10 elements.
2	<b>ArrayList(Collection&lt;? extends E&gt; c)</b> This constructor is used to create a list containing the elements of the specified collection.
3	<b>ArrayList(int initialCapacity)</b> This constructor is used to create an empty list with an initial capacity.

- Methods:

S.N.	Method & Description
1	<b>void add(int index, E element)</b> This method inserts the specified element at the specified position in this list.
2	<b>boolean addAll(Collection&lt;? extends E&gt; c)</b> This method appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator
3	<b>void clear()</b> This method removes all of the elements from this list.
4	<b>boolean contains(Object o)</b> This method returns true if this list contains the specified element.
5	<b>E get(int index)</b> This method returns the element at the specified position in this list.
6	<b>int indexOf(Object o)</b> This method returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

7	<b>boolean isEmpty()</b> This method returns true if this list contains no elements.
8	<b>int lastIndexOf(Object o)</b> This method returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
9	<b>boolean remove(Object o)</b> This method removes the first occurrence of the specified element from this list, if it is present.
10	<b>E set(int index, E element)</b> This method replaces the element at the specified position in this list with the specified element.
11	<b>int size()</b> This method returns the number of elements in this list.
12	<b>Object[] toArray()</b> This method returns an array containing all of the elements in this list in proper sequence (from first to last element).

## LinkedList Class.

- The LinkedList class extends AbstractSequentialList and implements the List interface. It provides a linked-list data structure.
- Constructors:

SN	Constructors and Description
1	<b>LinkedList( )</b> This constructor builds an empty linked list.
2	<b>LinkedList(Collection c)</b> This constructor builds a linked list that is initialized with the elements of the collection c.

- Methods:

SN	Methods with Description
1	<b>void add(int index, Object element)</b>

	<p>Inserts the specified element at the specified position index in this list. Throws <code>IndexOutOfBoundsException</code> if the specified index is out of range (<code>index &lt; 0    index &gt; size()</code>).</p>
2	<p><b>boolean addAll(Collection c)</b></p> <p>Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. Throws <code>NullPointerException</code> if the specified collection is null</p>
3	<p><b>void addFirst(Object o)</b></p> <p>Inserts the given element at the beginning of this list.</p>
4	<p><b>void addLast(Object o)</b></p> <p>Appends the given element to the end of this list.</p>
5	<p><b>void clear()</b></p> <p>Removes all of the elements from this list.</p>
6	<p><b>boolean contains(Object o)</b></p> <p>Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element <code>e</code> such that <code>(o==null ? e==null : o.equals(e))</code>.</p>
7	<p><b>Object get(int index)</b></p> <p>Returns the element at the specified position in this list. Throws <code>IndexOutOfBoundsException</code> if the specified index is out of range (<code>index &lt; 0    index &gt;= size()</code>).</p>
8	<p><b>Object getFirst()</b></p> <p>Returns the first element in this list. Throws <code>NoSuchElementException</code> if this list is empty.</p>
9	<p><b>Object getLast()</b></p> <p>Returns the last element in this list. Throws <code>NoSuchElementException</code> if this list is empty.</p>
10	<p><b>int indexOf(Object o)</b></p> <p>Returns the index in this list of the first occurrence of the specified element, or -1 if the List does not contain this element.</p>
11	<p><b>int lastIndexOf(Object o)</b></p> <p>Returns the index in this list of the last occurrence of the specified element, or -1 if the</p>

	list does not contain this element.
12	<b>Object remove(int index)</b> Removes the element at the specified position in this list. Throws NoSuchElementException if this list is empty.
13	<b>boolean remove(Object o)</b> Removes the first occurrence of the specified element in this list. Throws NoSuchElementException if this list is empty. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0    index >= size()).
14	<b>Object removeFirst()</b> Removes and returns the first element from this list. Throws NoSuchElementException if this list is empty.
15	<b>Object removeLast()</b> Removes and returns the last element from this list. Throws NoSuchElementException if this list is empty.
16	<b>Object set(int index, Object element)</b> Replaces the element at the specified position in this list with the specified element. Throws IndexOutOfBoundsException if the specified index is is out of range (index < 0    index >= size()).
17	<b>int size()</b> Returns the number of elements in this list.
18	<b>Object[] toArray()</b> Returns an array containing all of the elements in this list in the correct order. Throws NullPointerException if the specified array is null.

- Example :

```
import java.util.*;

public class LinkedListDemo {

    public static void main(String args[]) {
        // create a linked list
        LinkedList ll = new LinkedList();
        // add elements to the linked list
        ll.add("F");
        ll.add("B");
        ll.add("D");
    }
}
```



```

ll.add("E");
ll.add("C");
ll.addLast("Z");
ll.addFirst("A");
ll.add(1, "A2");
System.out.println("Original contents of ll: " + ll);

// remove elements from the linked list
ll.remove("F");
ll.remove(2);
System.out.println("Contents of ll after deletion: "
    + ll);

// remove first and last elements
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: "
    + ll);

// get and set a value
Object val = ll.get(2);
ll.set(2, (String) val + " Changed");
System.out.println("ll after change: " + ll);
    }
}

```

## Enumeration Interface.

- The Enumeration interface defines the methods by which you can enumerate (obtain one at a time) the elements in a collection of objects.
- This legacy interface has been superceded by Iterator. Although not deprecated, Enumeration is considered obsolete for new code. However, it is used by several methods defined by the legacy classes such as Vector and Properties, is used by several other API classes, and is currently in widespread use in application code.
- The methods declared by Enumeration are summarized in the following table:

SN	Methods with Description
1	<b>boolean hasMoreElements( )</b> When implemented, it must return true while there are still more elements to extract, and false when all the elements have been enumerated.
2	<b>Object nextElement( )</b> This returns the next object in the enumeration as a generic Object reference.

- Example:

```

import java.util.Vector;
import java.util.Enumeration;

```

```

public class EnumerationTester {

    public static void main(String args[]) {
        Enumeration days;
        Vector dayNames = new Vector();
        dayNames.add("Sunday");
        dayNames.add("Monday");
        dayNames.add("Tuesday");
        dayNames.add("Wednesday");
        dayNames.add("Thursday");
        dayNames.add("Friday");
        dayNames.add("Saturday");
        days = dayNames.elements();
        while (days.hasMoreElements()){
            System.out.println(days.nextElement());
        }
    }
}

```

## Vector Class.

- Vector implements a dynamic array. It is similar to ArrayList, but with two differences:
  - Vector is synchronized.
  - Vector contains many legacy methods that are not part of the collections framework.
- Vector proves to be very useful if you don't know the size of the array in advance or you just need one that can change sizes over the lifetime of a program.
- Constructors :

SR.NO	Constructor and Description
1	<b>Vector( )</b> This constructor creates a default vector, which has an initial size of 10
2	<b>Vector(int size)</b> This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size:
3	<b>Vector(int size, int incr)</b> This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward
4	<b>Vector(Collection c)</b> <b>creates a vector that contains the elements of collection c</b>

- Methods

SN	Methods with Description
1	<b>void add(int index, Object element)</b> Inserts the specified element at the specified position in this Vector.
2	<b>boolean addAll(Collection c)</b> Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
3	<b>void addElement(Object obj)</b> Adds the specified component to the end of this vector, increasing its size by one.
4	<b>int capacity()</b> Returns the current capacity of this vector.
5	<b>void clear()</b> Removes all of the elements from this Vector.
6	<b>boolean contains(Object elem)</b> Tests if the specified object is a component in this vector.
7	<b>boolean containsAll(Collection c)</b> Returns true if this Vector contains all of the elements in the specified Collection.
8	<b>Enumeration elements()</b> Returns an enumeration of the components of this vector.
9	<b>Object firstElement()</b> Returns the first component (the item at index 0) of this vector.
10	<b>Object get(int index)</b> Returns the element at the specified position in this Vector.
11	<b>int indexOf(Object elem)</b> Searches for the first occurrence of the given argument, testing for equality using the equals method.
12	<b>boolean isEmpty()</b> Tests if this vector has no components.
13	<b>Object lastElement()</b>

	Returns the last component of the vector.
14	<b>int lastIndexOf(Object elem)</b> Returns the index of the last occurrence of the specified object in this vector.
15	<b>Object remove(int index)</b> Removes the element at the specified position in this Vector.
16	<b>boolean removeAll(Collection c)</b> Removes from this Vector all of its elements that are contained in the specified Collection.
17	<b>Object set(int index, Object element)</b> Replaces the element at the specified position in this Vector with the specified element.
18	<b>int size()</b> Returns the number of components in this vector.

- Example:

```
import java.util.*;
public class VectorDemo {
    public static void main(String args[]) {
        // initial size is 3, increment is 2
        Vector v = new Vector(3, 2);
        System.out.println("Initial size: " + v.size());
        System.out.println("Initial capacity: " +
            v.capacity());
        v.addElement(new Integer(1));

        System.out.println("Capacity after additions: " +
            v.capacity());
        System.out.println("First element: " +
            (Integer)v.firstElement());
        System.out.println("Last element: " +
            (Integer)v.lastElement());
        if(v.contains(new Integer(3)))
            System.out.println("Vector contains 3.");
        // enumerate the elements in the vector.
        Enumeration vEnum = v.elements();
        System.out.println("\nElements in vector:");
        while(vEnum.hasMoreElements())
            System.out.print(vEnum.nextElement() + " ");
        System.out.println();
    }
}
```

## Properties Class.

- Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.
- The Properties class is used by many other Java classes. For example, it is the type of object returned by `System.getProperties()` when obtaining environmental values.
- Properties define the following instance variable. This variable holds a default property list associated with a Properties object.
- Constructors:

SR.No	Constructors and Description
1	<b>Properties()</b> This constructor creates a Properties object that has no default values
2	<b>Properties(Properties propDefault)</b> creates an object that uses propDefault for its default values. In both cases, the property list is empty

- Methods :

SN	Methods with Description
1	<b>String getProperty(String key)</b> Returns the value associated with key. A null object is returned if key is neither in the list nor in the default property list.
2	<b>String getProperty(String key, String defaultProperty)</b> Returns the value associated with key. defaultProperty is returned if key is neither in the list nor in the default property list.
3	<b>void list(PrintStream streamOut)</b> Sends the property list to the output stream linked to streamOut.
4	<b>void list(PrintWriter streamOut)</b> Sends the property list to the output stream linked to streamOut.
5	<b>void load(InputStream streamIn) throws IOException</b> Inputs a property list from the input stream linked to streamIn.
6	<b>Enumeration propertyNames()</b> Returns an enumeration of the keys. This includes those keys found in the default property list, too.
7	<b>Object setProperty(String key, String value)</b> Associates value with key. Returns the previous value associated with key, or returns null if no such association exists
8	<b>void store(OutputStream streamOut, String description)</b>

After writing the string specified by description, the property list is written to the output stream linked to streamOut

- Example :

```
import java.util.*;
public class PropDemo {
    public static void main(String args[]) {
        Properties capitals = new Properties();
        Set states;
        String str;

        capitals.put("Illinois", "Springfield");
        capitals.put("Missouri", "Jefferson City");
        capitals.put("Washington", "Olympia");
        capitals.put("California", "Sacramento");
        capitals.put("Indiana", "Indianapolis");

        // Show all states and capitals in hashtable.
        states = capitals.keySet(); // get set-view of keys
        Iterator itr = states.iterator();
        while(itr.hasNext()) {
            str = (String) itr.next();
            System.out.println("The capital of " +
                               str + " is " + capitals.getProperty(str) + ".");
        }
        System.out.println();

        // look for state not in list -- specify default
        str = capitals.getProperty("Florida", "Not Found");
        System.out.println("The capital of Florida is "
                           + str + ".");
    }
}
```

# InetAddress class

- Java InetAddress class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name for example www.gtu.ac.in, www.google.com, www.facebook.com etc.
- Methods :

SN	Methods with Description
1	<b>public static InetAddress getByName(String host) throws UnknownHostException</b> it returns the instance of InetAddress containing LocalHost IP and name.
2	<b>public static InetAddress getLocalHost() throws UnknownHostException</b> it returns the instance of InetAddress containing local host name and address.
3	<b>public String getHostName()</b> it returns the host name of the IP address.
4	<b>public String.getHostAddress()</b> it returns the IP address in string format.

- A sample of a List is given below:

```
import java.io.*;
import java.net.*;
public class InetDemo{
    public static void main(String[] args)
    {
        try{
            InetAddress ip=InetAddress.getByName("www.javatpoint.com");
            System.out.println("Host Name: "+ip.getHostName());
            System.out.println("IP Address: "+ip.getHostAddress());
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}
```

## Socket Class.

- This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines.
- The actual work of the socket is performed by an instance of the SocketImpl class. An application, by changing the socket factory that creates the socket implementation, can configure itself to create sockets appropriate to the local firewall.
- Constructors:

S.N.	Constructor & Description
1	<b>Socket()</b> Creates an unconnected socket, with the system-default type of SocketImpl.
2	<b>Socket(InetAddress address, int port)</b> Creates a stream socket and connects it to the specified port number at the specified IP address.

- Methods:

S.N.	Method & Description
1	<b>close()</b> Closes this socket.
2	<b>connect(SocketAddress endpoint)</b> Connects this socket to the server.
3	<b>getInetAddress()</b> Returns the address to which the socket is connected.
4	<b>getPort()</b> Returns the remote port number to which this socket is connected.
5	<b>isConnected()</b> Returns the connection state of the socket.
6	<b>getInputStream()</b> returns the InputStream attached with this socket.
7	<b>getOutputStream()</b> returns the OutputStream attached with this socket.



## ServerSocket Class.

- This class implements server sockets. A server socket waits for requests to come in over the network. It performs some operation based on that request, and then possibly returns a result to the requester.
- The actual work of the server socket is performed by an instance of the SocketImpl class. An application can change the socket factory that creates the socket implementation to configure itself to create sockets appropriate to the local firewall.
- Constructors:

SN	Constructors and Description
1	<b>ServerSocket()</b> This constructor builds an empty linked list.
2	<b>ServerSocket(int port)</b> Creates a server socket, bound to the specified port.

- Methods:

SN	Methods with Description
1	<b>accept()</b> Listens for a connection to be made to this socket and accepts it.
2	<b>close()</b> Closes this socket.
3	<b>getInetAddress()</b> Returns the local address of this server socket.

## Example of Java Socket Programming

- File: MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {
    public static void main(String[] args){
        try{
            ServerSocket ss=new ServerSocket(6666);
            Socket s=ss.accept();//establishes connection
            DataInputStream dis=new DataInputStream(s.getInputStream());
            String str=(String)dis.readUTF();
            System.out.println("message= "+str);
            ss.close();
        }
        catch(Exception e)
        {
```

```

        System.out.println(e);
    }
}

```

- File: MyClient.java

```

import java.io.*;
import java.net.*;
public class MyClient {

    public static void main(String[] args)
    {
        try
        {
            Socket s=new Socket("localhost",6666);
            DataOutputStream dout=new DataOutputStream(s.getOutputStream());
            dout.writeUTF("Hello Server");
            dout.flush();
            dout.close();
            s.close();
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
    }
}

```

## DatagramSocket Class & DatagramPacket Class

- **DatagramSocket :**
- Java DatagramSocket class represents a connection-less socket for sending and receiving datagram packets.
- A datagram is basically an information but there is no guarantee of its content, arrival or arrival time.
- The Commonly Used Constructors of DatagramSocket:

SN	Methods with Description
1	<b>DatagramSocket()</b> it creates a datagram socket and binds it with the available Port Number on the localhost machine.
2	<b>DatagramSocket(int port)</b> it creates a datagram socket and binds it with the given Port Number.

- The Commonly Used Methods of Datagram Socket:

SN	Methods with Description
1	<b>send(DatagramPacket p)</b> Sends a datagram packet from this socket.
2	<b>receive(DatagramPacket p)</b> Receives a datagram packet from this socket.

- **DatagramPacket:**

- Java DatagramPacket is a message that can be sent or received. If you send multiple packet, it may arrive in any order. Additionally, packet delivery is not guaranteed.
- The Commonly Used Constructors of DatagramPacket:

SN	Methods with Description
1	<b>DatagramPacket(byte[] barr, int length)</b> it creates a datagram packet. This constructor is used to receive the packets.
2	<b>DatagramPacket(byte[] barr, int length, InetAddress address, int port)</b> it creates a datagram packet. This constructor is used to send the packets.

- Example of Sending DatagramPacket by DatagramSocket:

```
//DSender.java
import java.net.*;
public class DSender{
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket();
        String str = "Welcome java";
        InetAddress ip = InetAddress.getByName("127.0.0.1");

        DatagramPacket dp = new DatagramPacket(str.getBytes(), str.length(), ip, 3000);
        ds.send(dp);
        ds.close();
    }
}
```

```
//DReceiver.java
import java.net.*;
public class DReceiver{
    public static void main(String[] args) throws Exception {
        DatagramSocket ds = new DatagramSocket(3000);
        byte[] buf = new byte[1024];
        DatagramPacket dp = new DatagramPacket(buf, 1024);
        ds.receive(dp);
        String str = new String(dp.getData(), 0, dp.getLength());
        System.out.println(str);
        ds.close();
    }
}
```

}

# Introduction to Object orientation

- Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts.
- The fundamental construct is object, which combine data structure and behavior.
- Object-Oriented Models are useful for
  - i. Understanding problems
  - ii. Communicating with application experts
  - iii. Modeling enterprises
  - iv. Preparing documentation
  - v. Designing programs and databases
- The subject is not primarily about OO-language or coding.
- It emphasize on initial stages of process development i.e. Requirement Gathering, planning etc. Object-oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects.

## Explain Object Orientation in detail.

- Object-oriented (OO) means that we organize software as a collection of isolated objects that incorporate both data structure and behavior.
- Object Orientation is about viewing and modeling the world/system as a set of interacting and interrelated objects.
- Characteristics of OO approach include four aspects: identity, classification, inheritance and polymorphism.

### i. Identity

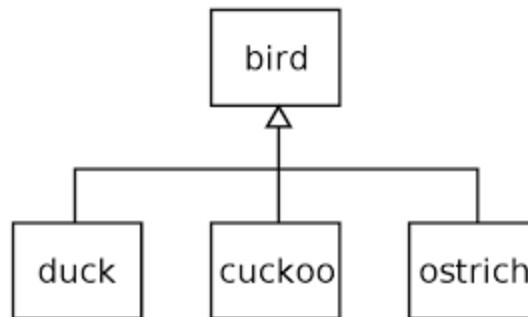
- Identity means data is quantized into discrete and distinguishable entities. Each object is having its own inherent identity.  
e.g. identity of Pen object is very different with an identity of Table object
- Objects are distinct even if all their attribute values (i.e. name and size) are identical. In programming language each object has unique handle by which it can be referenced.
- Objects can be concrete (real/existing), such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocessing operating system.
- Language implements the handle in various ways, such as an address, array index, etc.

### ii. Classification

- Abstract entities with the same structure (attributes) and behavior (operations) are grouped into classes. Example: Paragraph, Chess Piece etc.
  - A class is an abstraction that describes properties important to an application and ignores the rest.
  - Each class describes a possibly infinite set of individual objects.
  - Each object is said to be an instance of its class.
  - An object has its own value for each attribute but shares the attribute names and operations with other instances of the class.
  - Choice of class is arbitrary and depends on application.
  - Each class describe infinite set of individual object and each object is instance of class.
-

### iii. Inheritance

- Sharing of attributes and operations based on a hierarchical relationship. Each subclass inherits all features of super class and adds its unique features.
- A super class has general information that subclasses refine and elaborate. Each subclass incorporates, or inherits all the features of its super class and adds its own unique features.
- Subclasses need not repeat the features of the super class.  
Example, Scrolling Window, and Fixed Window are subclasses of Window.
- The ability to factor out common features of several classes into a super class can greatly reduce repetition within designs and programs and is one of the main.



**Figure : Inheritance**

### iv. Polymorphism

- Polymorphism means that the same operation may behave differently for different classes. Example, the move operation behaves differently for a pawn than for the queen in a chess game.
- Polymorphism- a Greek term means ability to take more than one form.
- The same operation may behave differently on different classes
- An operation is a procedure or transformation that an object performs or is subject to.
- An implementation of an operation by a specific class is called a method.
- An Operation is a procedure or transformation that an object performs.
- Implementation of operation by a specific class is called method.

## Modeling as a Design Technique and modeling concepts

### Abstraction

- Abstraction is fundamental human capability that permits us to deal with complexity.
- A process allowing focusing on most important aspects while ignoring less important details. Abstraction is the selective examination of certain aspects of a problem.
- The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant.
- Abstraction determines what is, and is not, important.
- The purpose of an abstraction is to limit the universe so we can understand. A good model captures the crucial aspects of a problem and omits the others.
- In building model we must not search for absolute truth but for adequacy.
- There is no single “correct” model, only adequate and inadequate ones.

## Property of Abstraction:-

- All the abstraction are incomplete and inaccurate.
- No Abstraction is perfect.  
E.g. All human words and language are abstractions (incomplete description of real world).
- No single model is sufficient to represent any situation.
- Abstraction does not destroy usefulness of a system.
- The purpose is to limit the universe which eliminates the complexity.
- Software model that contains extra detail unnecessarily limits choice of design decisions and diverts from the real issues.

## Explain Three models

### Class Model

- The structure of an object in system includes:
  - i. Identity
  - ii. Relationship to other object
  - iii. Attributes
  - iv. Operations
- This model provides context for state and interaction model.
- The goal in constructing class model is to capture those concepts from real world that are important to application.

### State Model

- Describes history and behavior of a system with respect to time.
- Explains context of events and organization of events with states.
- Actions and events in state model become operations on object in class model.

### Interaction Model

- Explains interactions between objects.
- It also explains, how individual object collaborate to achieve behavior of system as a whole.
  - i. **Use Case Model:** Document major theme for interactions between system and outside actors.
  - ii. **Sequence Model:** Describes object that interact and time sequence of their interactions.
  - iii. **Activity Model:** shows flow of control among processing steps of a computation.

### Relationship among the Models

- Each model describes one aspect of the system but contains references to the other models, The class model describes data structure on which the state and interaction models operate.
  - The operations in the class model correspond to events and actions. The state model describes the control structure of objects. It shows decision that depends on object values causes actions that change object values and state. The interaction models focuses on the exchanges between objects and provide a complete overview of the operation of a system.
-

## Class Modeling

- A class modeling captures static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and operations for each class of objects.
- Class model is most important among three models.
- Emphasizes on building a system around objects rather than functionality.
- Class Model closely corresponds to the real world and is consequently more flexible with respect to the change.
- The purpose of class modeling is to describe object.

## Object and Class Concepts

### Object:

- Object is a concept, abstraction, or a thing with identity that has meaning for an application  
E.g. Two apples each have identity and are distinguishable.
- Objects are instances of classes.
- It often appears as a proper nouns or specific references in problem descriptions
- Some objects have real world counterparts (name of a person/company).
- While some object have conceptual entity (formula for solving quadratic equation).
- Choice of object depends on judgment and the nature of a problem. There can be many correct representations. All Object have identity and distinguishable.
- Identity means objects are distinguished by their inherent existence and not by descriptive properties.
- Real-world objects share two characteristics: They all have attributes and behavior.

### Class:

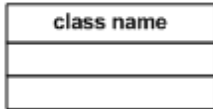
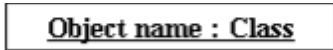
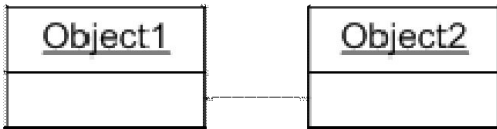
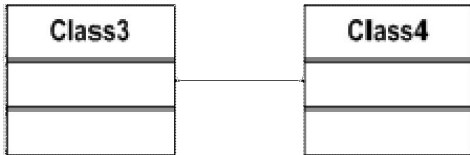

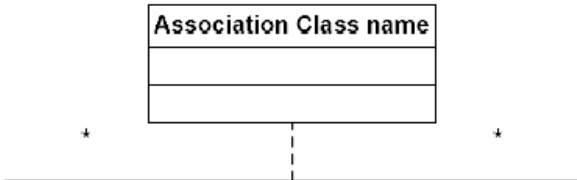
- Class describes a group of objects with the same properties (attributes), behaviour (operations), kinds of relationship, and Semantics. (E.g.: Person, Company, Process and Window).
- A software unit that implements one or more interfaces.
- Classes often appear as common noun and noun phrase in problem description.
- By grouping objects into class, we abstract a problem.

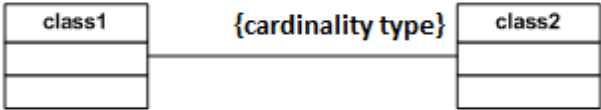
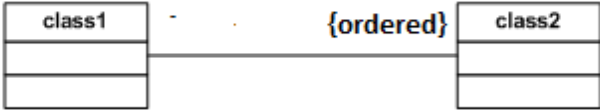

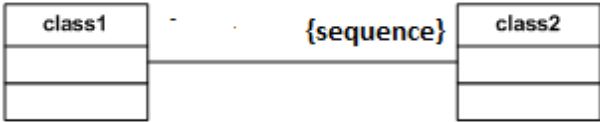
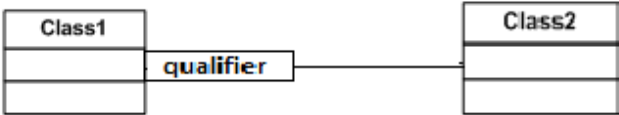
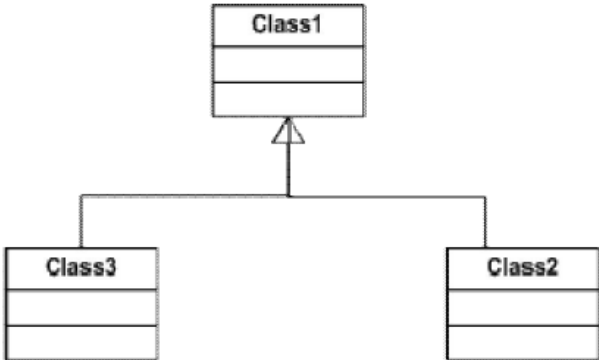
## Class Diagram

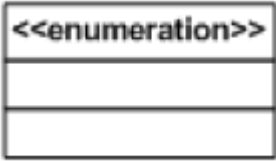
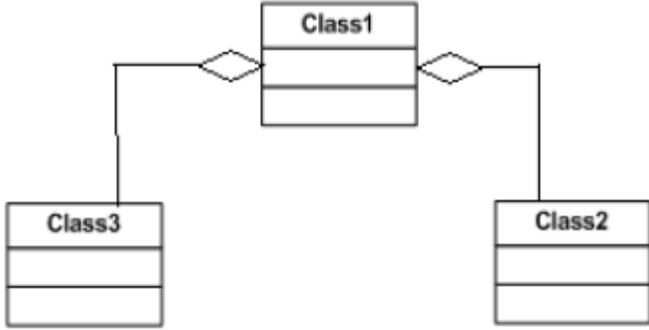

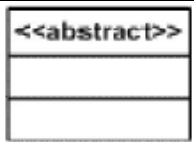
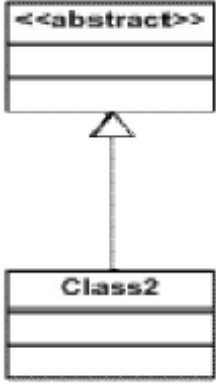
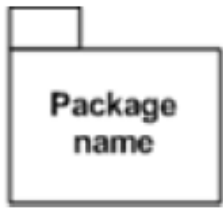
- Class Diagram provides a Graphical notation for modeling classes and their relationships, thereby describing possible objects.
  - The most widely used diagram of UML.
  - Models the static design view of a system.
  - Useful in modeling business objects.
  - Used to specify the structure, interfaces and relationships between classes that underlie the system architecture.
  - Primary diagram for generating codes from UML models.
-



## Class Diagram Notations

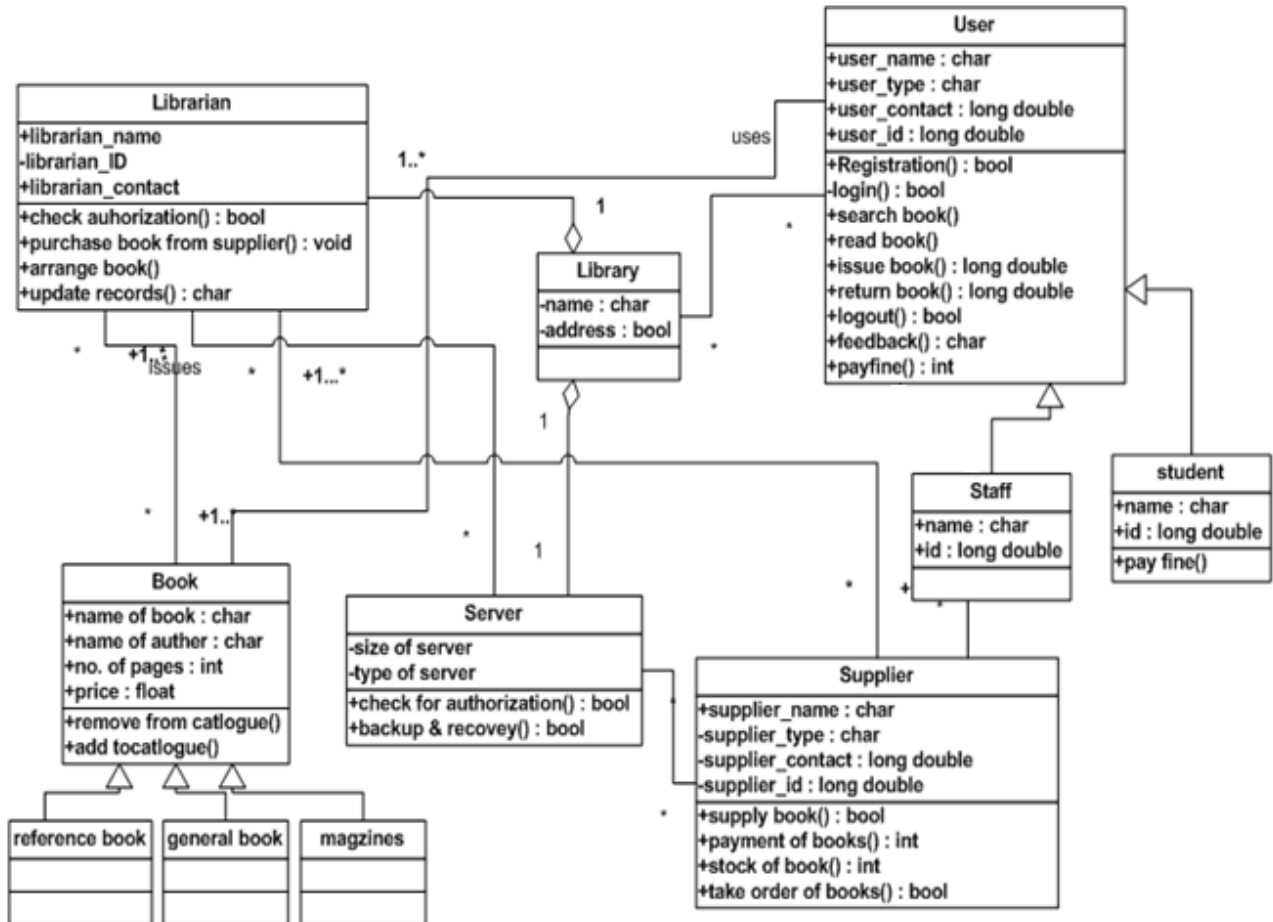
Sr. No.	Name	Symbol	Meaning
1	Class		Class is an entity of the class diagram. It describes a group of objects with same properties & behavior.
2	Object		An object is an instance or occurrence of a class.
3	Link		A link is a physical or conceptual connection among objects.
4	Association		An association is a description of a links with common structure & common semantics.
5	Multiplicity	<p>Ex. 1    to    1                  1    to    *                  *    to    *                  *    to    1                  1    to    0...2</p> 	Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. It is a constraint on the Cardinlity of a set.
6	Association class		It is an association that is a class which describes the association with attributes.

7	cardinality		It describes the count of elements from collection.
8	ordering		It is used to indicate an ordered set of objects with no duplication allowed.
9	bag		A bag is a collection of unordered elements with Duplicates allowed.
10	sequence		A sequence is an ordered collection of elements with duplicates allowed.
11	qualified association		Qualification increases the precision of a model. It is used to avoid many to many multiplicities and it converts into one to one multiplicity.
12	generalization		Generalization organizes classes by their superclass and sub-class relationship.

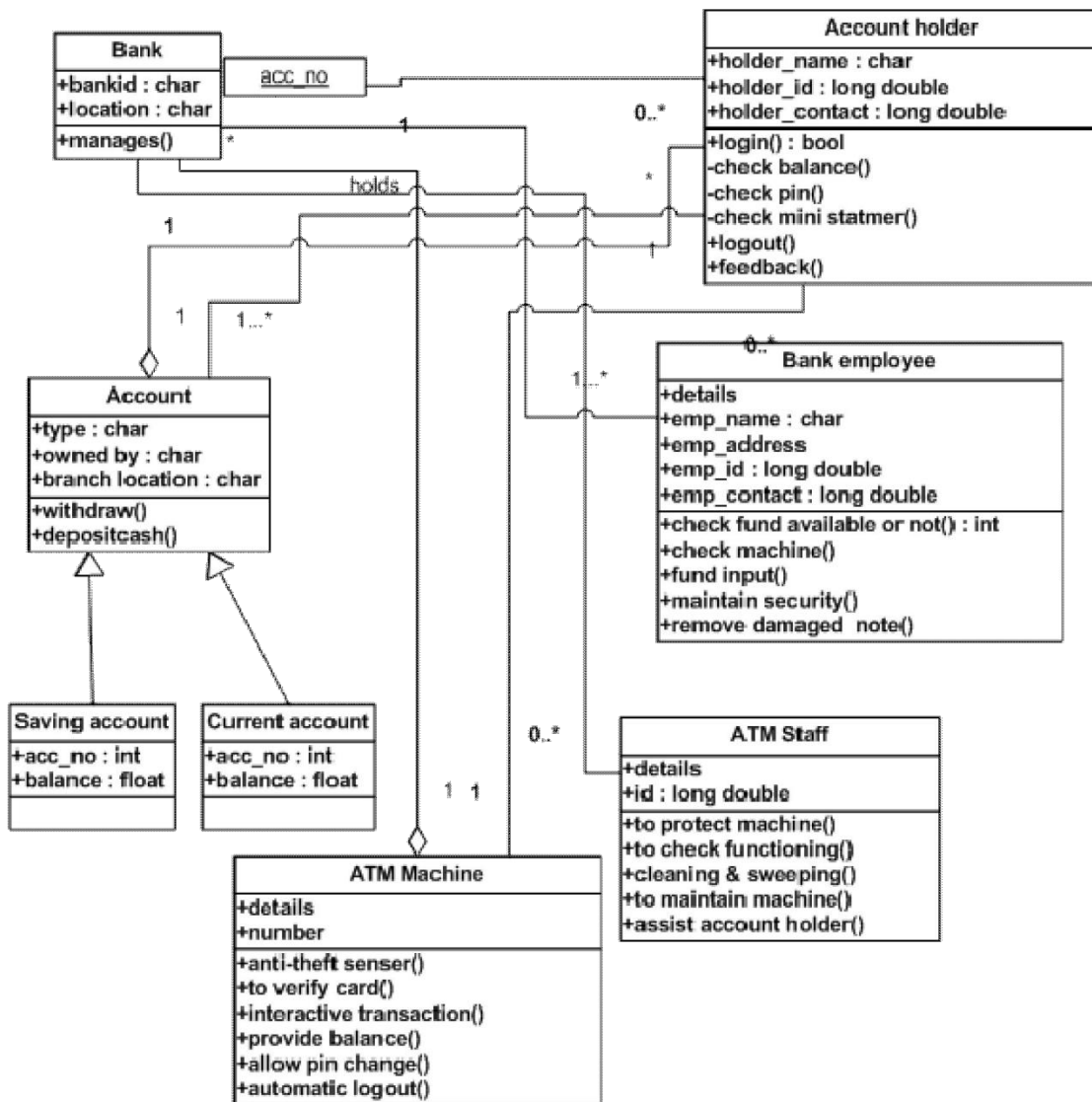
13	Enumeration		An enumeration is a data type that has a finite set of values.
14	Aggregation		It is a strong form of association in which an aggregate object is made of constituent parts.
15	Composition		It is a form of aggregation. Composition implies ownership of the parts by the whole.
16	Abstract class		It is a class that has no direct instances.
17	Concrete class		It is a class that is intangible; it can have direct instances. Class-2 is example of concrete class
18	package		A package is a group of elements with common theme.

## Examples:

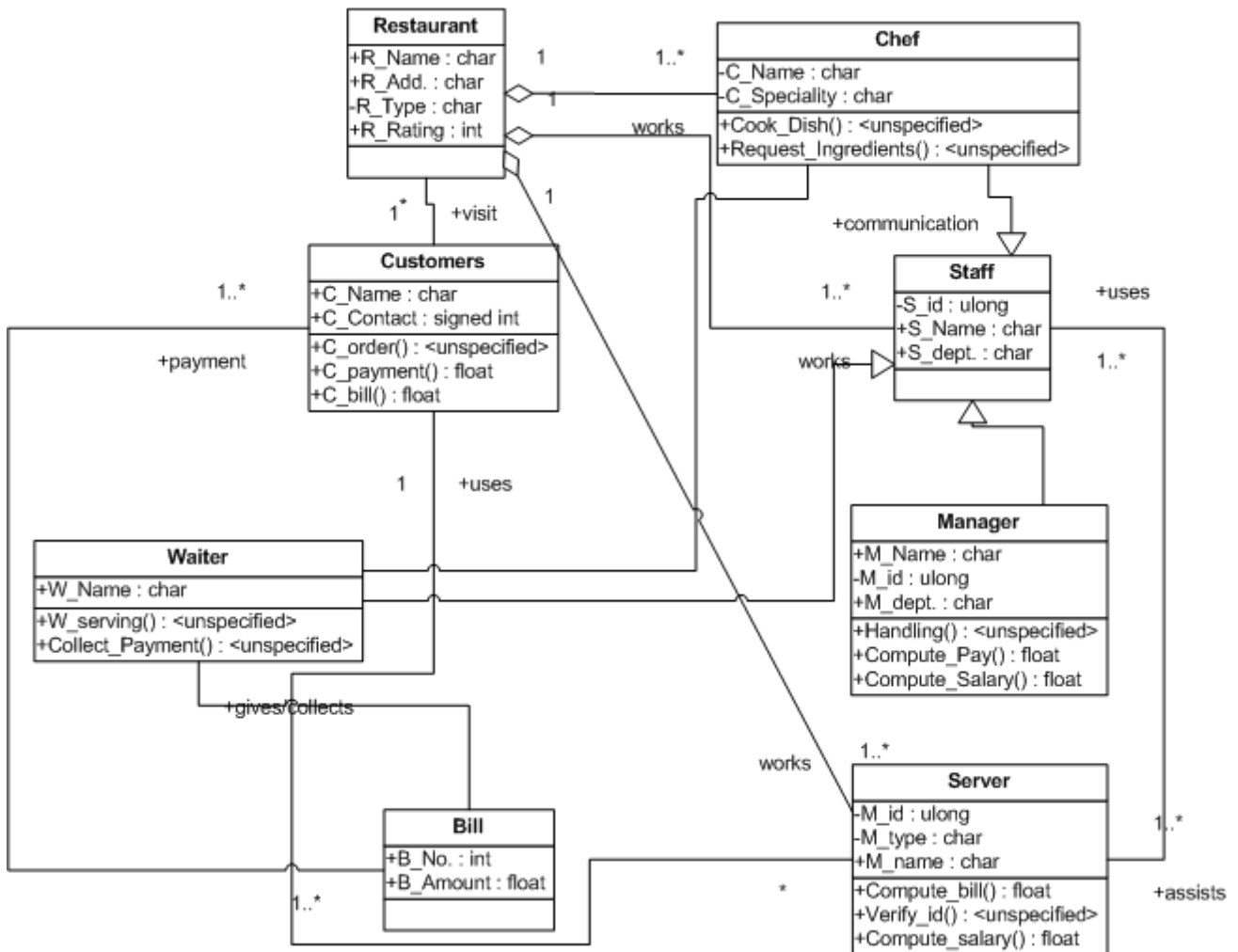
### Class Diagram for Library Management System



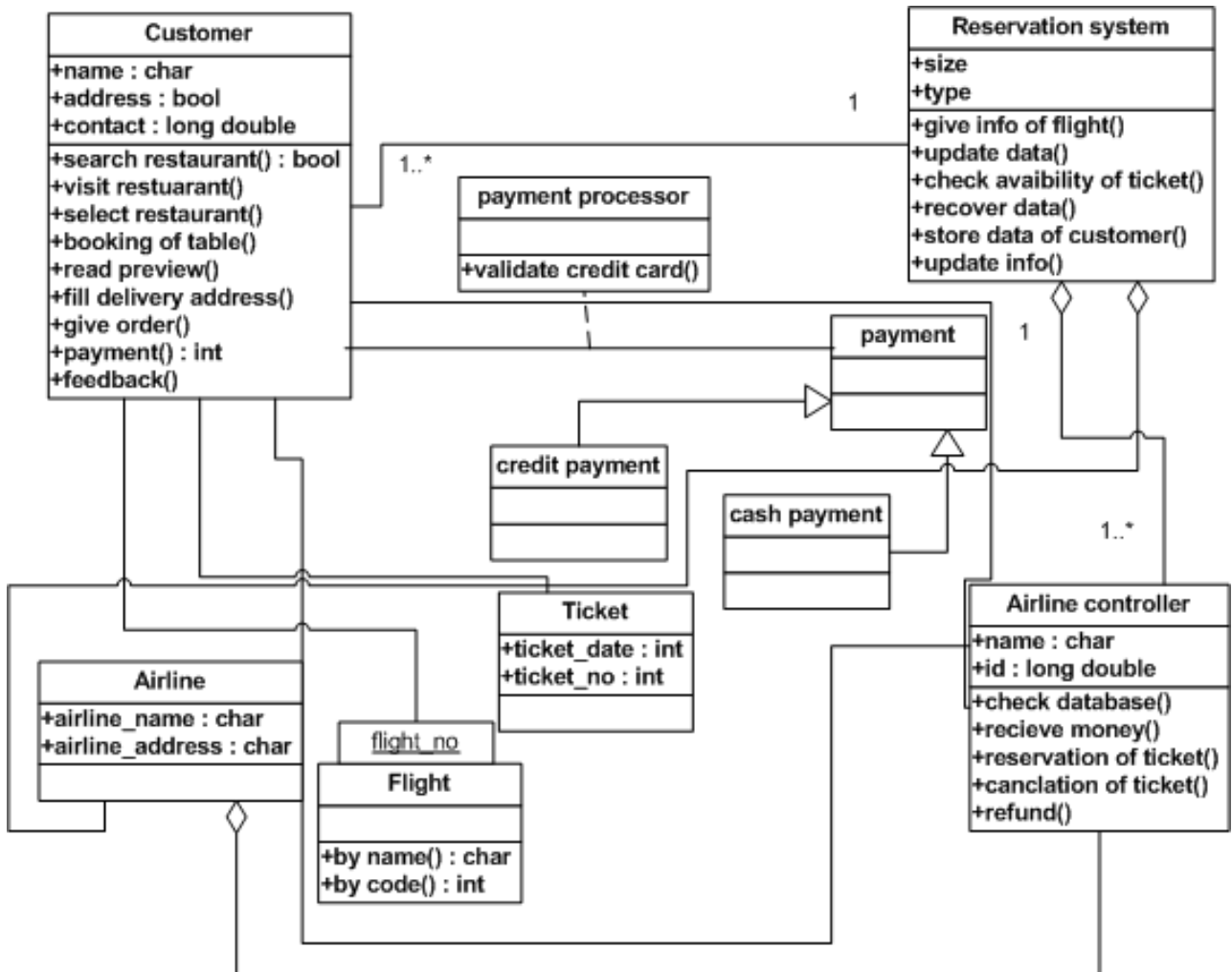
## Class Diagram for ATM



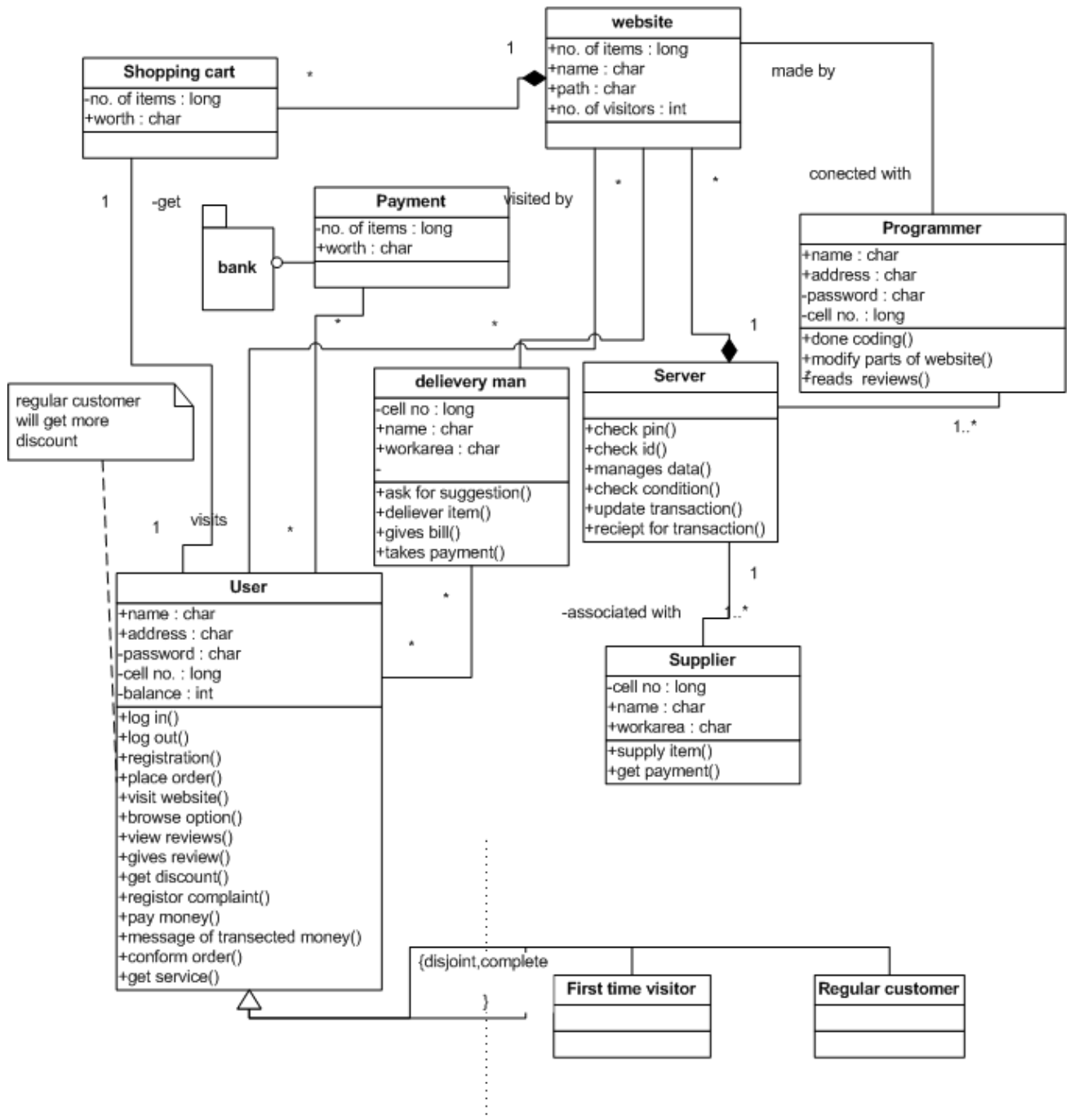
## Class Diagram for Online Restaurant System



## Class Diagram for Online Reservation System



## Class Diagram for Online Shopping System



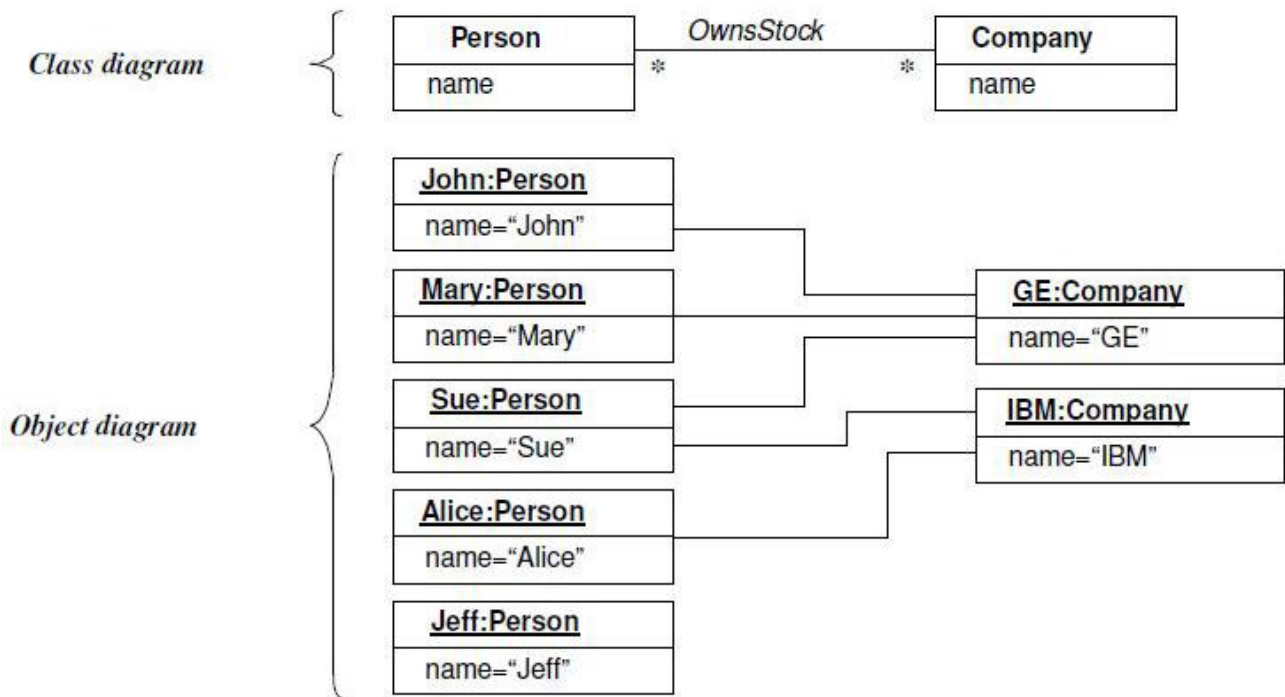


## Purpose of class diagram

- Analysis and design of the static view of an application
- Describes responsibilities of a system.
- Base for component and deployment diagrams.

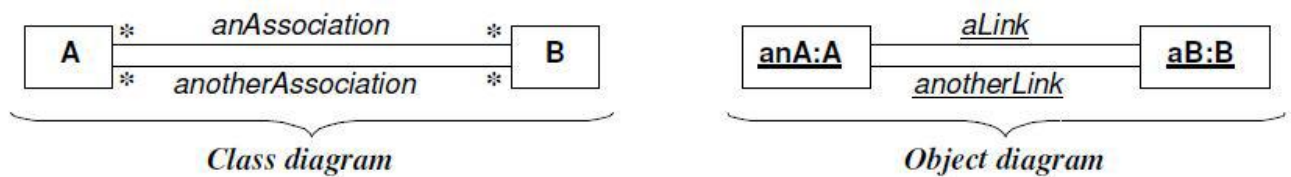
## Link and Association

- Link and Association are the means for establishing relationship among objects and classes.
- Link and Association often appears as verbs in problem statement.
- Link is a physical / conceptual connection among objects most links relate two objects, but some links relate three or more object. It is an instance of association as shown in figure below.
- Association is a description of a group of links with common structure and common semantics as in the class diagram shown in figure below.



**Figure: Many-to-many Association**

- Associations are indirectly bidirectional.
- Both the direction of traversal is equally meaningful.
- It is the only name of the association that establishes the direction.
- Developers often implements associations in programming language as a references from one object to another.
- Associations are important, precisely because they break encapsulation.
- Associations cannot be private to a class, because they go beyond classes.



**Figure: Association Vs. Link**

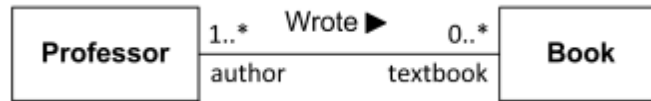
## Generalization and Inheritance

- Generalization is the relationship between a class (the super class) and one or more variations of the class (sub class).
- Super class holds the common attributes, operations and associations. Subclass adds specific attributes.
- Each subclass inherits features of super class Ancestor and descendents.
- Use of Generalization serves three purposes:
  - i. Support for polymorphism. (call at super class level automatically resolved)
  - ii. Second purpose is to structure the description of objects. (a taxonomy is formed)
  - iii. Third purpose is to enable reuse of code.
- The terms generalization, specialization and inheritance all refer to aspects of the same idea.
- Generalization: derives from the fact that the sub class generalizes to super class
- Specialization: refers to the fact that the subclasses refine or specialize the super-class.
- Inheritance: Is the mechanism for sharing attributes, operations, and associations via generalization specialization relationship which is useful for parent child relationship.

## Advance class Modeling

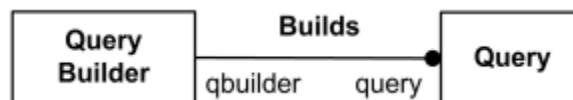
### Advance object and class concepts and Association Ends.

- Association end is a connection between the line depicting an association and the icon depicting the connected classifier.
- Name of the association end may be placed near the end of the line. The association end name is commonly referred to as role name (but it is not defined as such in the UML 2.4 standard).
- The role name is optional and suppressible.



*Professor "playing the role" of author is associated with textbook end typed as Book.*

- - The idea of the role is that the same classifier can play the same or different roles in other associations.
- - For example, Professor could be an author of some Books or an editor.
- Association end could be owned either by end classifier, or association itself
- Association ends of associations with more than two ends must be owned by the association. Ownership of association ends by an associated classifier may be indicated graphically by a small filled circle (aka dot).
- The dot is drawn at the point where line meets the classifier.
- It could be interpreted as showing that the model includes a property of the type represented by the classifier touched by the dot.
- This property is owned by the classifier at the other end.

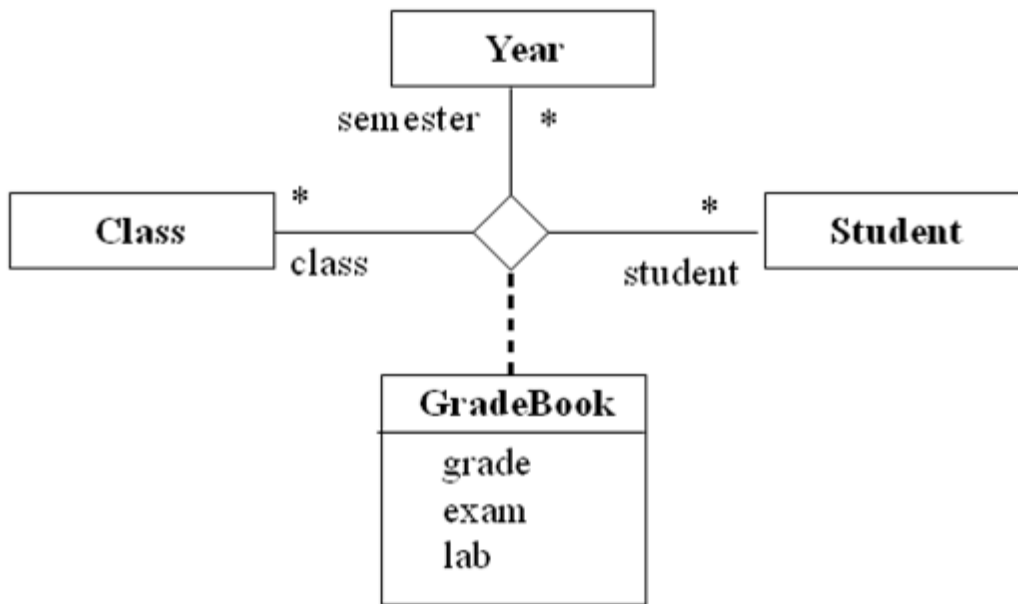


*Association end query is owned by classifier QueryBuilder and association end qbuilder is owned by association Builds itself*

- The "ownership" dot may be used in combination with the other graphic line-path notations for properties of associations and association ends. These include aggregation type and navigability.

### N-ary associations/ Ternary Association

- N-ary association means associations among three or more classes.
- A ternary association is an association with three roles that cannot be restated as binary associations.
- The notation for a ternary association is a large diamond; each associated class connects to a vertex of the diamond with a line.

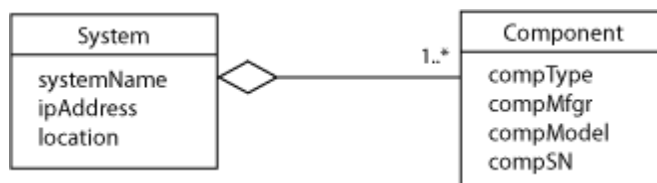


- The above figure represents another example of Ternary Association.
- You should try to avoid n-ary associations.
- Normally it is decomposed into binary associations with possible qualifiers and attributes.
- Many relationships involve just two things and can be modeled with the simple binary association.
- It is not however uncommon for three or more things to be involved in a relationship.
- An n-ary association can be used in these circumstances and allows any or "n" number of things to be related in a single cohesive group.
- An n-ary association is used when the three or more things are all related to each other in a structural or behavioral way.
- It does not replace the use of two binary associations where a classifier is related to two other classifiers, but the latter two classifiers aren't related to each other.
- In below Figure a professor teaches a listed course for a semester.
- The delivered course may use many textbooks; the same textbook may be used for multiple delivered courses.

## Aggregation and Composition

### Aggregation

- Aggregation is a strong form of association in which an aggregate object is made of constituent parts.
- An aggregation as relating an assembly class to one constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregations.
- Aggregation is a special form of association.



- If two objects are tightly bound by a part-whole relationship, it is an aggregation.
- Aggregation is drawn like association; except a small diamond indicates the assembly end.
- The UML has two forms of part-whole relationships.
- A general form called aggregation
- A more restrictive form called composition.

#### Property of an Aggregation:-

- i. **Transitivity:** If A is part of B and B is part of C, then A is part of C. Aggregation
- ii. **Anti-symmetric:** If A is part of B, then B is not part of A.

### Composition

- Composition is a restricted form of aggregation with two additional constraints.
  - i. A constituent part can belong to at most one assembly.
  - ii. Once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly.

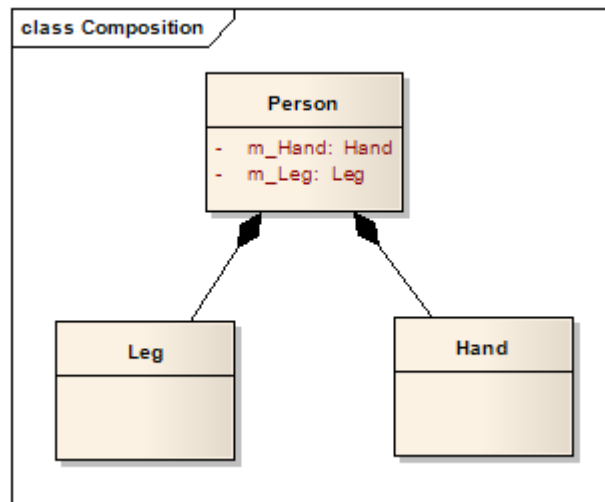
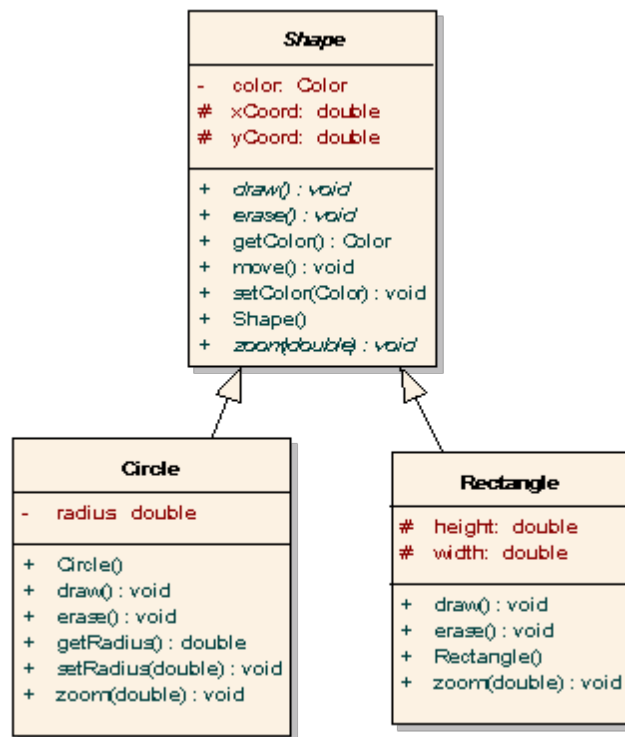


Figure of Composition

### Abstract Class

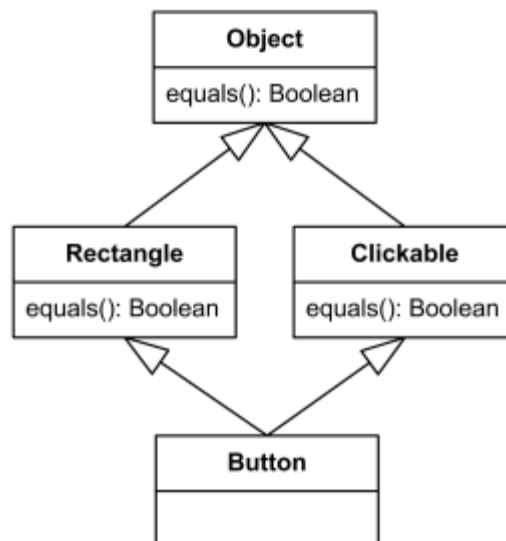
- An abstract class is a class that has no direct instances but whose descendant classes have direct instances.
- A concrete class is a class in which it can have direct instances.
- Abstraction is a process to allow focusing on most important aspects while ignoring less important details.
- In the UML notation an abstract class name is listed in an italic font. Or you may place the keyword {abstract} below or after the name.
- Use abstract class to define the signature for an operation without supplying a corresponding method.
- An abstract operation defines the signature of an operation for which each concrete subclass must provide its own implementation.
- As shown in example draw () is an abstract operation.
- Within abstract class (Graphic Object), draw () is just a definition and not implementation.

- Each subclass (Circle and Rectangle) must apply method draw () in its implementation. In other words all the super class are abstract class all the subclass are concrete class.
- It is advisable to avoid concrete super class.
- We can eliminate concrete super class by introducing other class. Differentiate Abstract class and Concrete class



## Multiple Inheritance

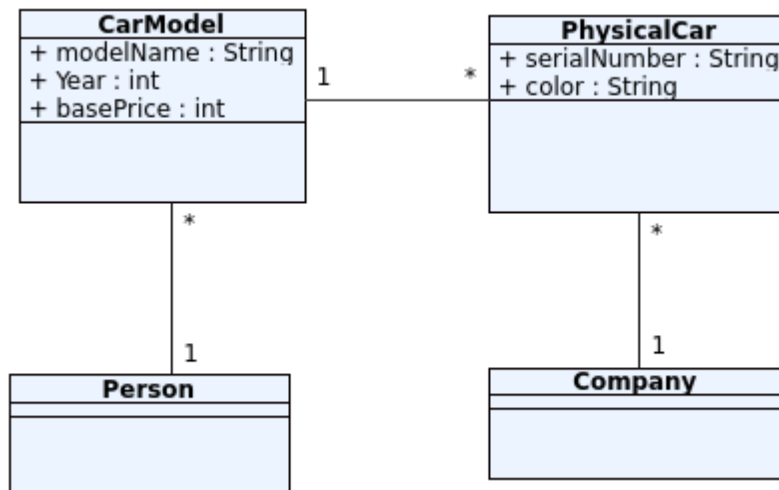
- Multiple Inheritance permits the class to have more than one super class
- Here subclass inherit feature from its all super class.



- In the multiple inheritance diamond problem example above Button class inherits two different implementations of equals() while it has no own implementation of the operation.
- When button.equals() is called, it is unknown which implementation from Rectangle or from Clickable will be used.
- It may arise conflicts among parallel definition creates ambiguities that implementation must resolve.

## Meta Data

- Metadata is data that describes other data.
- Data about data.
- All the UML software models are inherently metadata, since describe the thing being modeled.



**Figure: Metadata Example**

- Many real world applications have metadata.
- Computer-language implementations also use metadata heavily.
- We can consider classes as an object, but classes are meta-objects and not real-world object.
- Class description objects have features, and they in turn have their own classes, which are called Meta classes.
- Treating everything as an object provides a more uniform implementation. Metadata provides greater functionality for solving complex problem.
- Accessibility of metadata varies from language to language.
- Some language access metadata at compile time and some at run time.

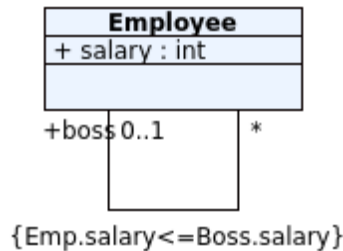
## Constraints in class modeling.

- A constraint is a Boolean condition involving model elements, such as objects, classes, attributes, links, associations and generalization sets.
- A constraint restricts the values that elements can assume.
- We can express constraint with

i. Natural Language

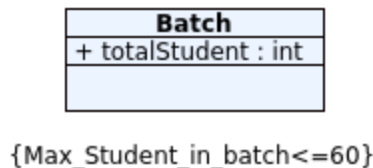
ii. Formal Language such as Object Constraint Language (OCL)

- Constraints on object is helpful to add explicit constraint on the object of a class.



**Figure: Constraint on object**

- As shown in above example, represents that no employee's salary can exceed salary of employee Boss (A constraint between two things at same time).
- Another example, for maximum student in a Batch.



**Figure: Constraint on object**

### Constraints on Generalization

- The semantics of generalization imply certain structural constraints.
- With the single inheritance the subclass is mutually exclusive.
- Furthermore, each instance of an abstract super class corresponds to exactly one subclass instance and each instance of a concrete super class corresponds to at most one subclass instance.
- UML defines certain keywords to demonstrate constraint.

#### i. Disjoint:

- The subclasses are mutually exclusive.
- Each object of subclass belongs to exactly one of the subclasses.



## ii. Overlapping:

- In an overlapping specialization, an individual of the parent class may be a member of more than one of the specialized subclasses.
- The subclasses can share some objects.
- An object may belong to more than one subclass.

## iii. Complete:

- Generalization that lists all possible subclasses.

## iv. Incomplete:

- Generalization in which some of the subclasses is missing.

## v. Static

- Generalization in which subclass are static in nature.

## vi. Dynamic

- Generalization in which subclass are dynamic with respect to time.

## Constraints on Links

- Multiplicity is constraint on cardinality of set.
- Multiplicity restricts number of object related to given object.
- Qualification also adds constraint on an association.
- An association class has a constraint that an ordinary class does not have; i.e. it derives identity from instance of related classes.

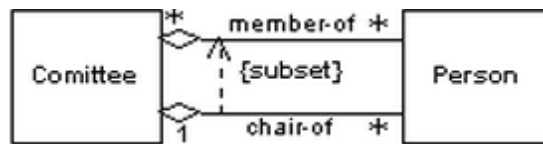
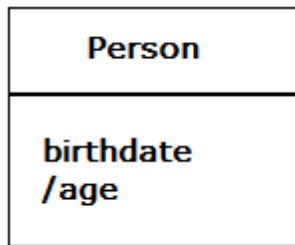


Figure: Subset constraint between association

- We favor expressing constraint in declarative manner in UML class diagram.
- If we don't use constraint in UML class modeling then we need to convert it into procedural form before the implementation; rather this is the straight forward way.
- Constraint provides criteria for measuring quality of class model.
- "A good software model" captures many constraints thought its structure.
- It often requires several iterations to get structure of a model right from the perspective of constraint.
- In practice we can't enforce every constraint, but should try to enforce the important ones.

## Derived Data.

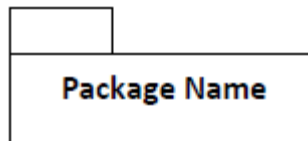
- A derived element is a function of one or more elements, which in turn may be derived.
- A derived element is redundant, because other element completely determines it.
- Classes, association and attribute may be derived.
- The notation for derived elements is a slash (/) in front of element name.



**{age=currentDate - birthdate}**

## Packages.

- A package is a group of elements with a common theme.
- A package partitions a model, making it easier to understand and manage.
- Large applications may require several tiers of packages.
- The notation for a package is a box with a tab. The purpose of the tab is to suggest the enclosed contents.



## State Modeling

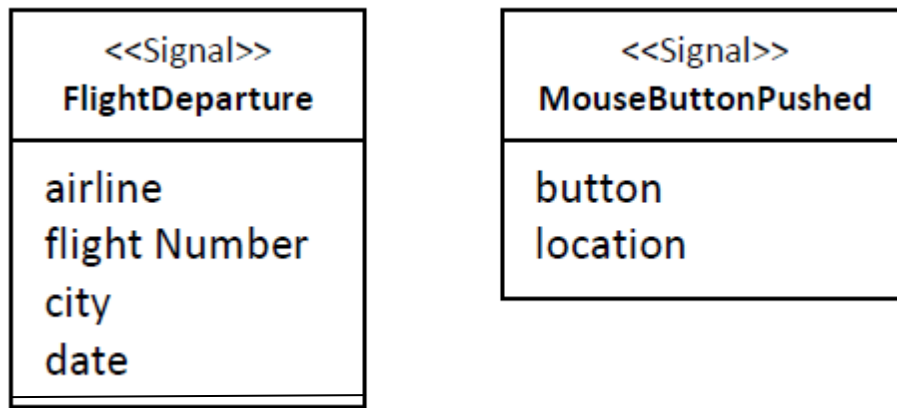
- State Diagram explains behavior of the system.
- A State Diagram is a graph whose nodes are states and arcs are transition between the states caused by the event.
- State Modeling examines changes to the object and their relationship over time.
- The behavior of an entity is not only a direct consequence of its input, but it also depends on its preceding state.
- The history of an entity can best be modeled by a finite state diagram.
- State diagram can show the different states of an entity also how an entity responds to various events by changing from one state to another.
- The major dynamic modeling concepts are events, which represent external stimuli, and states, which represent values of objects.
- State Diagram is a standard computer science concept/a graphical representation that relates events and states.
- State model describes:
  - i. Sequence of operation that occur in response of external stimuli.
  - ii. What the specific operation do.
  - iii. What they operate on.
  - iv. How that operation are implemented

## Events and types of Events in detail.

- Every event is a unique occurrence at a point in time.
- It causes transitions between states.
- Events might be related or unrelated. If two events are casually unrelated they are said to be concurrent.
- Event often corresponds to verb in past tense. i.e. door\_opened, door\_closed event.
- An event happens instantaneously with regard to the time scale of an application.
- It is simply an occurrence that application considers as atomic and short-lived.
- Events include error conditions like motor jammed, transaction aborted, and timeout, etc.
- In software modeling system we do not try to establish an ordering between concurrent events because they can occur in any order.
- The most common type of events is the signal event, the change event, and the time event.

## Signal Event

- Grouping every event into event classes and gives each event class a name to indicate common structure and behaviour is known as Signal Event.
  - It is a one-way transmission of information from one object to another. It is event of sending or receiving signal.
  - The UML notation is the keyword signal in guillemets (<< >>) above the signal class name in the top section of a box.
  - The difference between signal and signal event is :
    - **Signal:** one way transmission between object.
    - **Signal event:** an occurrence in time.
-



**Figure: Signal Event**

### Change Event

- Change event is caused by the satisfaction of Boolean expression. Whenever the expression changes from false to true the event happens.
- The UML notation for a change event is the keyword when followed by a parenthesized Boolean expression.

### Time Events

- A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.
- The UML notation for an absolute time is the keyword when followed by a parenthesized expression involving time.
- This event is caused by the occurrence of an absolute time or the elapsed of a time interval.
- The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to time duration.

### Explain State

- A state is an abstraction of the attribute values and links of an object.
- The response of an object to an event may include an action or a change of state by the object.
- A state corresponds to the interval between two events revived by an object. Events represent points in time; states represent intervals of time.
- A state has duration; it occupies an interval of time.
- A state is often associated with the value of an object satisfying some condition.
- In the simplest case, each enumerated value of an attribute defines a separate state.

### UML notation for state:

- A rounded box containing a state name.
- In defining state we ignore attributes that do not affect the behavior of the object.
- The convention is to list the state name in Bold Face, Centered and First Letter of the State Name should be CAPITALIZED
- The UML notation for a state – a rounded box containing a state name.

### Comparing Event and State

- Though Event and States having certain symmetry between them there exists some difference.
- Event represents points in time, while states represent intervals of time.

- The State having suggestive name and natural language description of its purpose.
- Write the characteristics of a state Alarm Ringing.
- The below figure shows for the state Alarm Ringing on a watch.

<b>State:</b> <i>Alarm ringing</i>		
<b>Description:</b> alarm on watch is ringing to indicate target time		
<b>Event sequence that produces the state:</b>		
<i>set alarm</i> (target time)		
any sequence not including <i>clear alarm</i>		
current time = target time		
<b>Condition that characterizes the state:</b>		
alarm = on, and $\text{target time} \leq \text{current time} \leq \text{target time} + 20 \text{ seconds}$ ,		
and no button has not been pushed since target time		
<b>Events accepted in the state:</b>		
<b>event</b>	<b>action</b>	<b>next state</b>
current time = target time + 20	reset alarm	<i>normal</i>
<i>button pushed</i> (any button)	reset alarm	<i>normal</i>

Figure: Various characterizations of a state

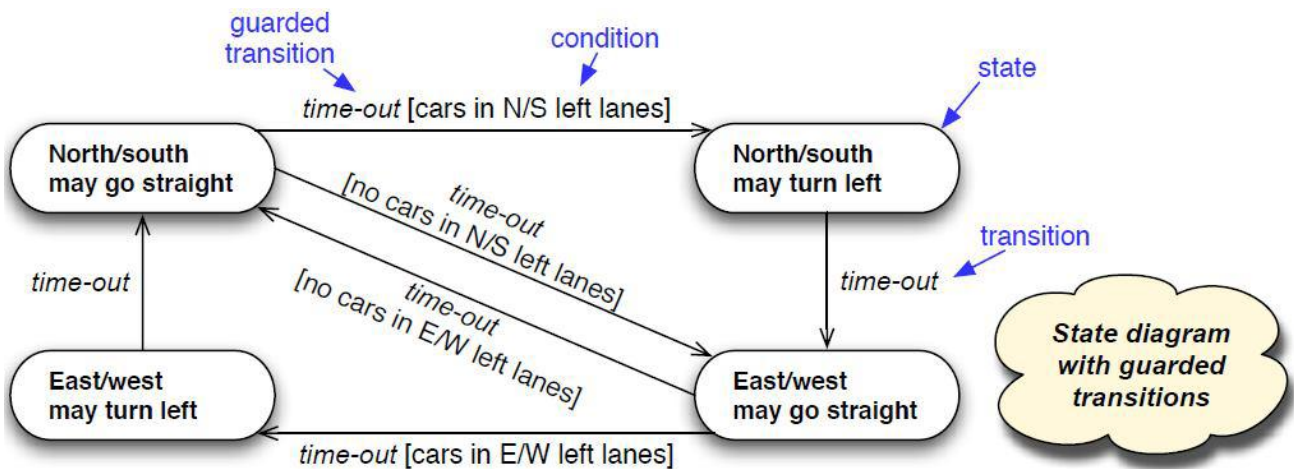
## Transition and Condition

### Transition

- A transition is a change from one state to another.
- A transition is a relationship between two states indicating that an object in the first state will perform certain actions and enter the second state when a specified event occurs and specified conditions are satisfied.
- On such a change of state, the transition is said to fire.
- Until the transition fires, the object is said to be in the source state; after it fires, it is said to be in the target state.
- Transition is an instantaneous change from one state to another.  
E.g. when a called phone is answered, phone line transition from *ringing state* to the *connected state*.

### Guard Condition

- Guard condition is a Boolean expression that must be true in order for a transition to occur.  
E.g. Traffic light is an intersection may change if road has cars waiting.
- Guard condition is the condition fired when event occurs, but only if guard condition is true.
- Guard condition is checked only once, at a time event occurs and transition fires if condition is true.
- If the condition is true later on, the transition does not fire then.
- UML notation for Guard condition is condition written inside square brackets [ ] on transition and followed by an event.
- It is an optional condition.



**Figure: Guarded Transitions**

### Comparing Guard Condition and Change Event

- Guard condition is checked only once while change event is, in effect and checked continuously.
- Guard condition is constraint, while Change event is a type of an event.

### State Diagram and their types.

#### State Diagram

- A State Diagram is a graph whose nodes are states and arcs are transition between the states caused by the event.
- State diagram specifies state sequence caused by an event sequence.
- State name must be unique within the scope of state diagram.
- A state diagram relates events and states. A change of state caused by an event is called transition. All the transitions leaving a state must correspond to different events.
- The state diagram specifies the state sequence caused by an event sequence.
- If an object is in a state and an event labelling one of its transitions occurs, the object.
- Enters the state on the target end of the transition. The transition is said to fire.
- A sequence of events corresponds to a path through the graph.
- There are two types of state diagram
  - i. Sample State Diagrams
  - ii. One-Shot State Diagrams

#### Sample State Diagrams

- State diagram with continuous loop.
- Sample state diagram represents object with infinite life.

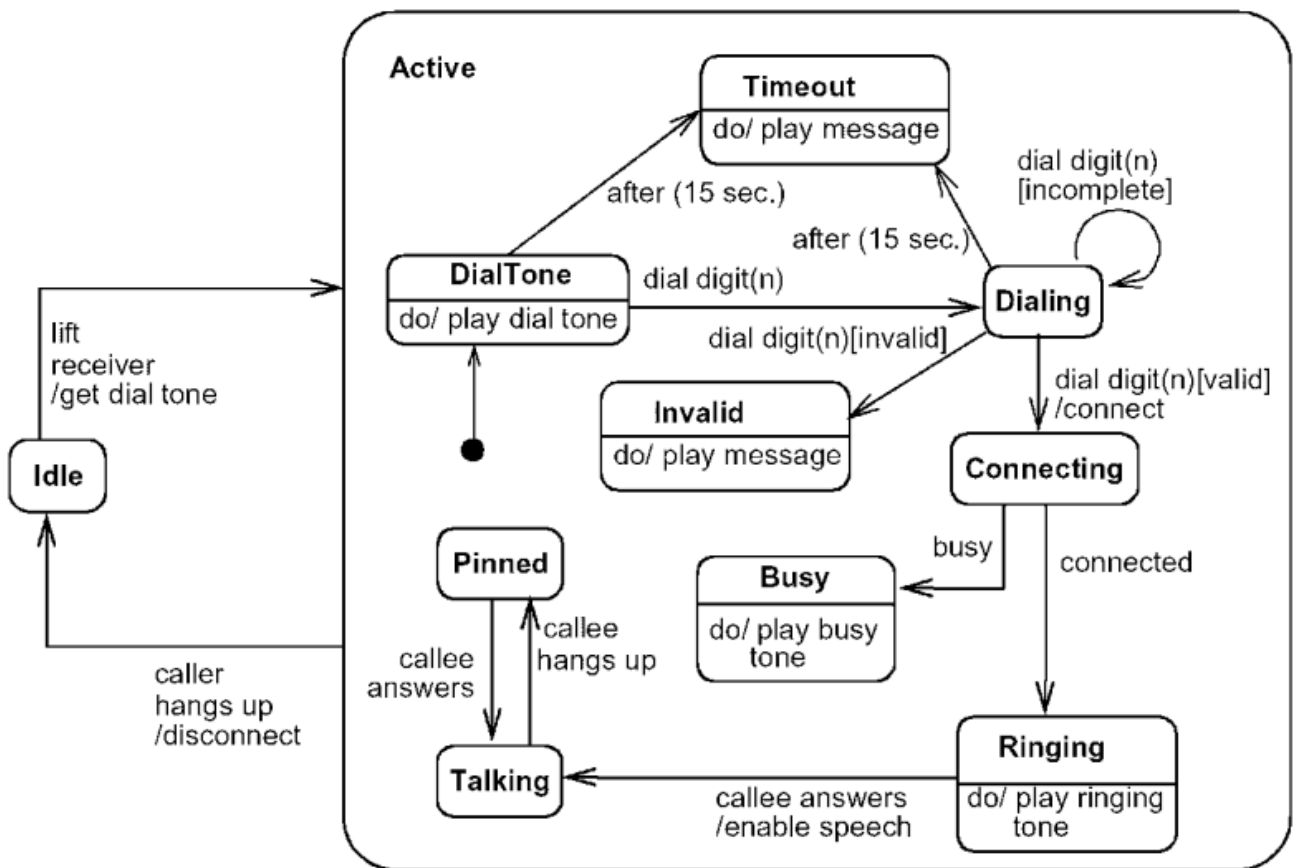


Figure: Sample State Diagram for Telephone Line

### One-Shot State Diagrams

- One-shot diagram represent object with finite lives and have initial and final states.
- Initial state is entered on creation of an object while entry of final state implies destruction of an object. You can indicate initial and final states via entry and exit points.
- Entry points (hollow circles) and exit points (circle enclosing an "x") appear on the state diagram's perimeter and may be named.

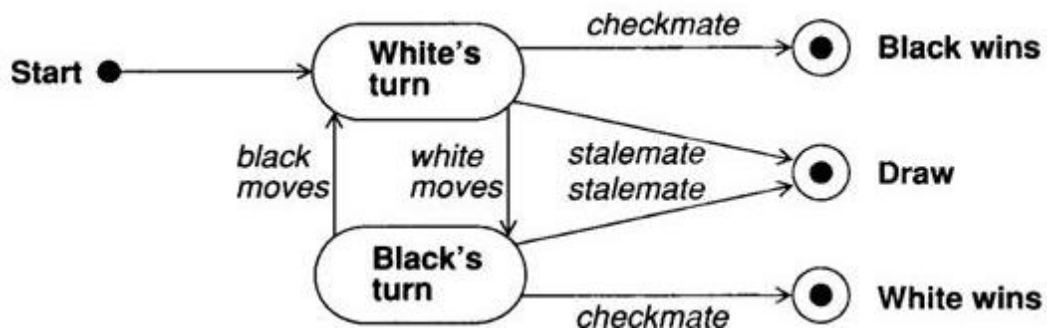


Figure: One shot State Diagram for Chess Game

### State Diagram Behavior

- Application of State diagram is not just limited to the description of events.
- A full description of an object must specify what the object does in response to events.

## Activity Effects

- The effect is a reference to a behavior that is executed in response to an event.
- An activity is the actual behavior that can be invoked by any number of effects.  
E.g. disconnect Phone Line might be an activity that is executed in response to hang up event.
- Activities can also represent internal control operations, such as setting attributes or generating other events.
- Such activities have no real-world counterparts but instead are mechanisms for structuring control within an implementation.  
E.g. program might increment an internal counter every time when specific event occurs.
- State diagram activity is denoted as slash ("/") and name or description of the activity, following the event that causes it.
- As shown in figure below, when right button is pressed, menu is displayed when it is released, menu is erased. While menu is visible, the highlighted menu item is updated whenever cursor moves.

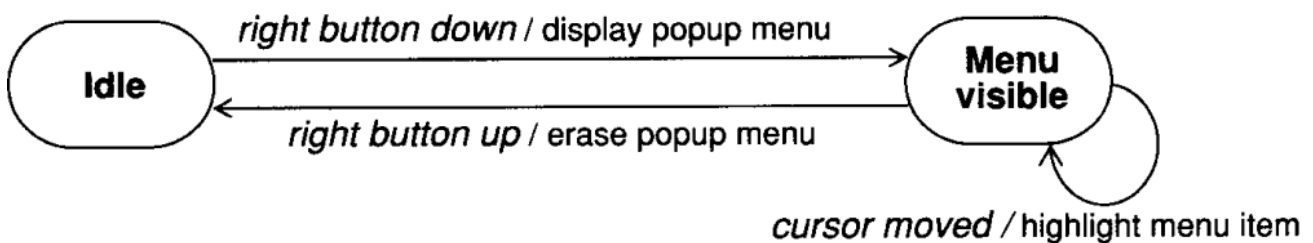


Figure: Activity for pop-up menu

## Do-activity

- UML notation for Do-activity is denoted as "do/" for all or part of duration that an object is in a state.
- It can only occur within a state and can't be attached to a transition.
- Do-activity may be interrupted by an event that is received during its execution. Such event may or may not cause a transition out of the state containing the do-activity.

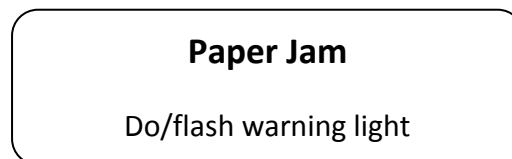
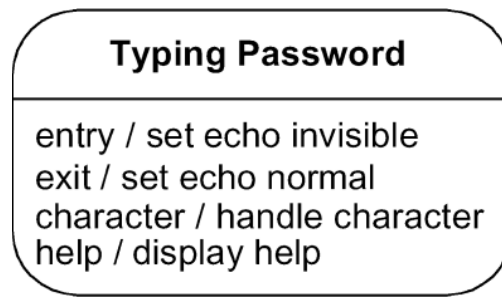


Figure: Do-activity for copy machine

## Entry and Exit Activities

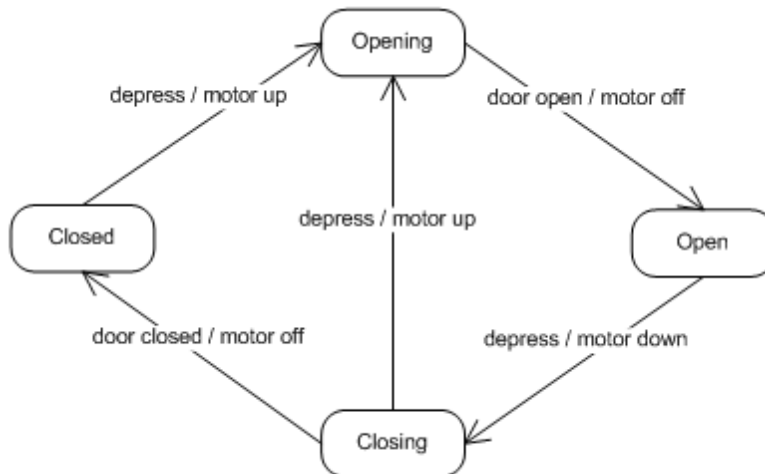
- We can also bind activities to entry or to exit from a state as an alternative to show the activity on transition.
- There is no difference between binding an activity to a transition or to a state.



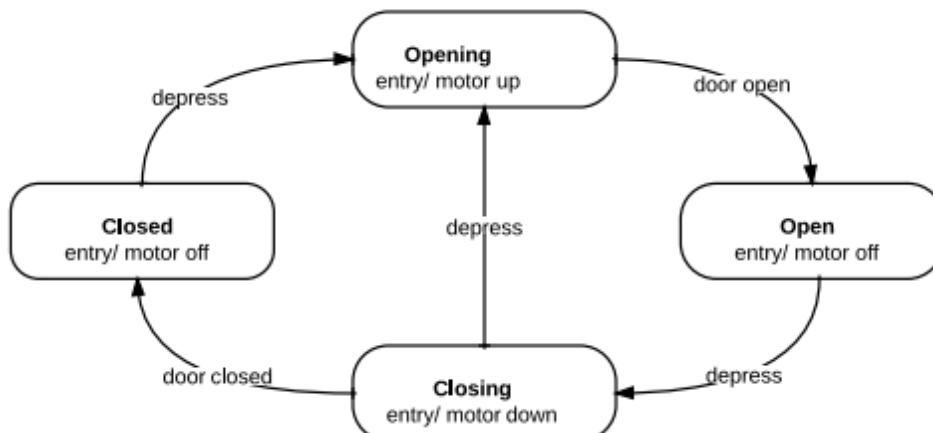


**Figure: Entry-Exit activities**

- Activity which is more concise to attach the activity of a state should be specified inside the state.
- When the state is entered by an incoming transition, entry activity is performed.
- An entry activity is equivalent to attaching the activity to every incoming transition.
- If incoming transition already has an activity, it is performed first.
- Exit activities are less common compared entry activities and occasionally useful.
- Whenever state is exited, by any outgoing transition, the exit activity is performed first.



**Figure: Activities on Transitions**



**Figure: Activities on entry to states**

- This diagram shows the states of door controller. User generates depress event with a pushbutton to open and close the door. Each event reverses the direction of the door. For safety door must open fully before it can be closed. Motor Up and Motor Down are activities which generates the control.
- Motor generates door open and door closed events when the motion has been completed.
- If a state has multiple activities, they are performed in following order:
  - 1: Activities on incoming transition
  - 2: Entry Activities
  - 3: Do-Activities
  - 4: Exit Activities
  - 5: Activities on outgoing transition
- If a Do-Activity is interrupted by transition out of the state, the exit activity is still performed.
- Therefore any event can occur within a state can cause an activity to be performed.

### Completion Transition

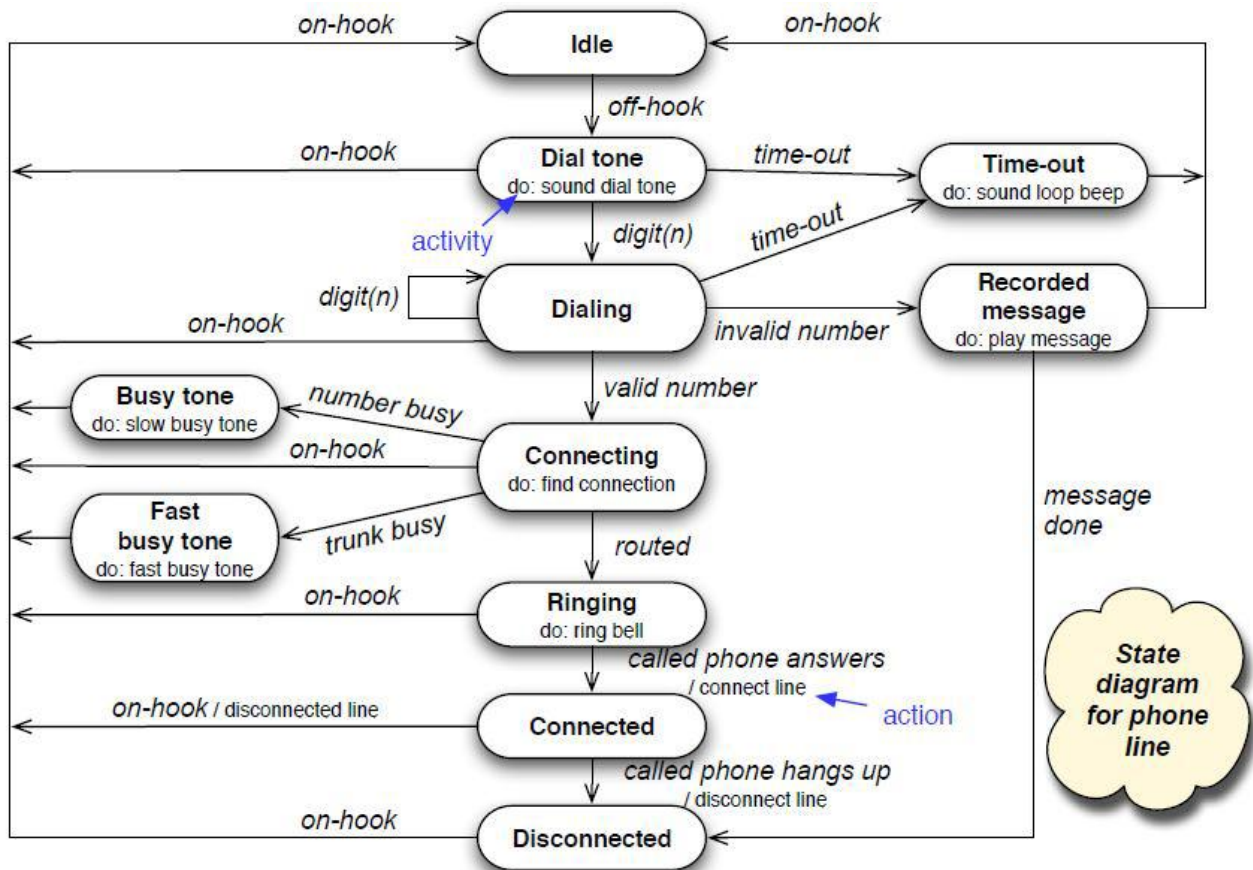
- Main purpose of state is to perform sequential activity.
- When an activity is completed, transition fires to another state.
- An arrow without an event name indicates an automatic transition that fires when activity associated with the source state is completed.
- Such unlabeled transition is known as Completion Transition.
- Stuck Condition: Guard condition is tested only once, when event occurs. If a state has one or more completion transition, but none of the guard condition is satisfied, then the state remains active and may become “stuck”.
- The completion event does not occur second time, therefore no completion transition will fire later to change the state.
- If the state has completion transition leaving it, normally the guard condition should cover every possible outcome.
- We can use some special condition such as else to apply if all other condition is false.
- Better approach is not to use guard condition on a completion transition, instead change event should be used.

### Sending Signals

- Object can perform activity by sending signal and system of objects interacts by exchanging signals.
- The activity “send target.S (attributes)” send signal S with given attributes to the target object or objects.

E.g. phone line sends connect (phone number) signal to the switcher when complete number has been dialed.
- A signal can be directed at a set of object or single object.
- If target is set of objects, each of them receives separate copy of signal and each of them independently processes the signal.
- **Race condition:** If object can receive signals from more than one object, the order in which concurrent signals are received may affect the final state. This is known as race condition.

The below diagram explain state diagram for telephone line with activities:



## Interaction Modeling

- Interaction model is the third leg of the tripod and describes interaction with the system.
- Class model describes object in the system and relationship among them.
- State model describes life history of an object.
- Interaction model describes how the object interacts to produce useful results.
- Interaction model and state software model describes whole behavior of the system.
- Interaction can be modeled at different level of abstraction. At higher level, use case describes how a system interacts with outside actors.
- Each use case represents piece of functionality that a system provides to its user. Use cases are helpful for capturing informal requirements.
- Sequence diagram provides more detail and show the messages exchanged among a set of objects over the time.
- Sequence diagrams are good for showing the behavior sequence seen by users of a system.
- Activity diagrams provide further detail and show the flow of control among the steps of computation.
- Activity diagram documents the steps necessary to implement an operation or a business process in a sequence diagram.

## Use case Models

- Use Cases identifies the functionality of a system and organize it according to the perspective of users.
- Use Cases describe complete transactions and are therefore less likely to omit necessary steps.
  - i. First determine the system
  - ii. Ensure that actors are focused
  - iii. Each use case must provide value to users.
  - iv. For example, dial a telephone number is not a good use case for a telephone system
  - v. Relate use cases and actors
  - vi. Remember that use cases are informal
  - vii. Use cases can be structured

## Use Case Diagram

### Introduction:

- A use case diagram describes how a system interacts with outside actors.
  - It is a graphical representation of the interaction among the elements and system.
  - Each use case representation a piece of functionality that a system provides to its user.
  - Use case identifies the functionality of a system.
  - Use case diagram allows for the specification of higher level user goals that the system must carry out.
  - These goals are not necessarily to tasks or actions, but can be more general required functionality of the system.
  - You can apply use case to capture the intended behavior of the system you are developing, without having to specify how that behavior is implemented.
-

- A use case diagram at its simplest is a representation of a user's interaction with the system and depicting the specifications of a use case.
- A use case diagram contains four components.
  - i. The boundary, which defines the system of interest in relation to the world around it.
  - ii. The actors, usually individuals involved with the system defined according to their roles.
  - iii. The use cases, which the specific roles are played by the actors within and around the system.
  - iv. The relationships between and among the actors and the use cases.

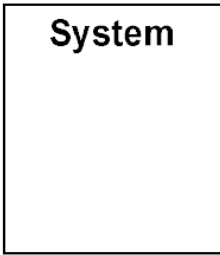
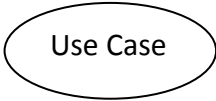

### Purpose:



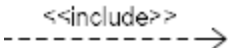
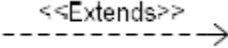
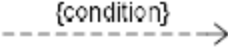
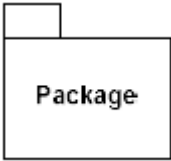



- The main purpose of the use case diagram is to capture the dynamic aspect of a system.
- Use case diagram shows, what software is suppose to do from user point of view.
- It describes the behavior of system from user's point.
- It provides functional description of system and its major processes.
- Use case diagram defines the scope of the system you are building.

### When to Use: Use Cases Diagrams

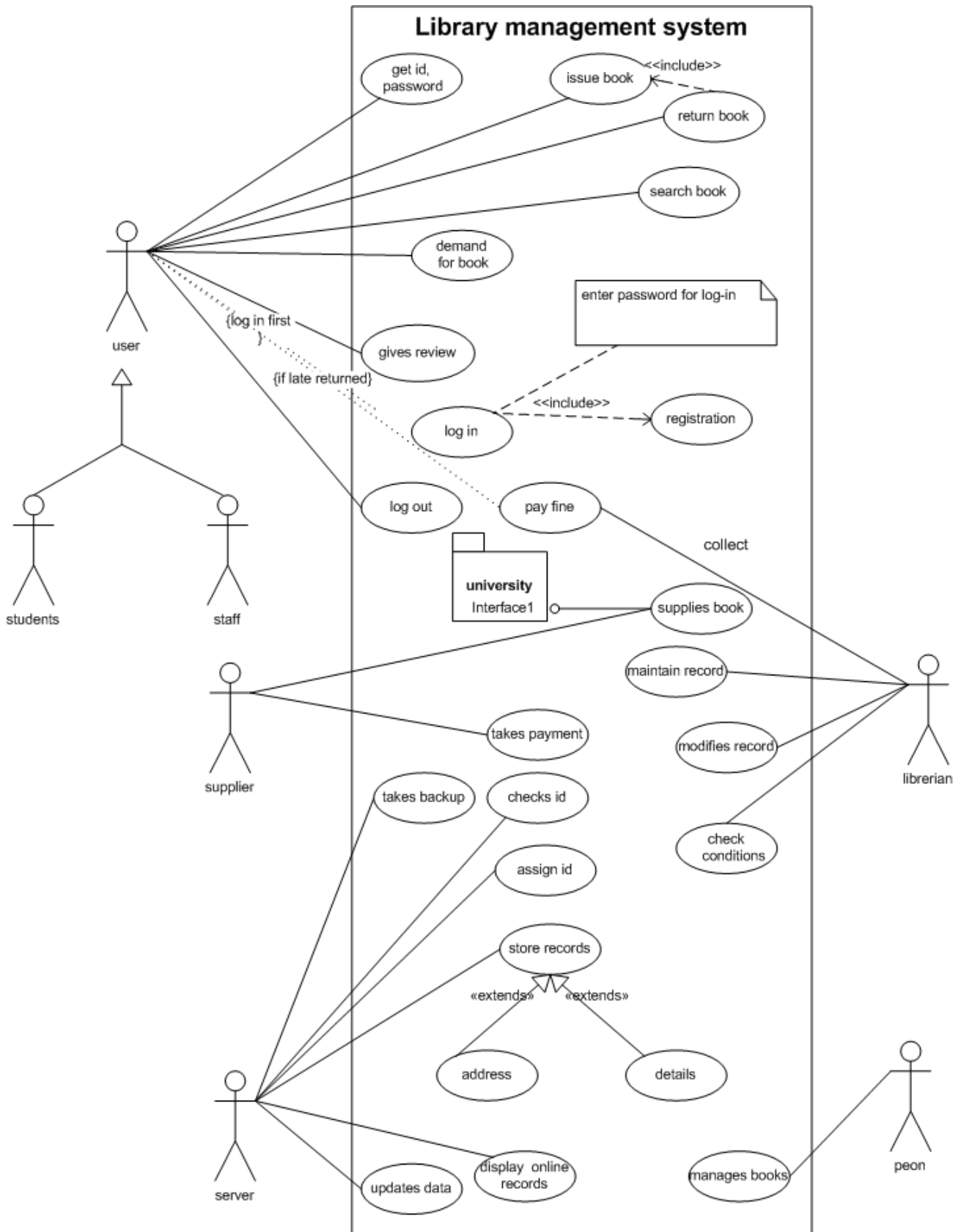
- Use cases are used in almost every project.
- They are helpful in exposing requirements and planning the project.
- During the initial stage of a project most use cases should be defined.

### Use Case Notations

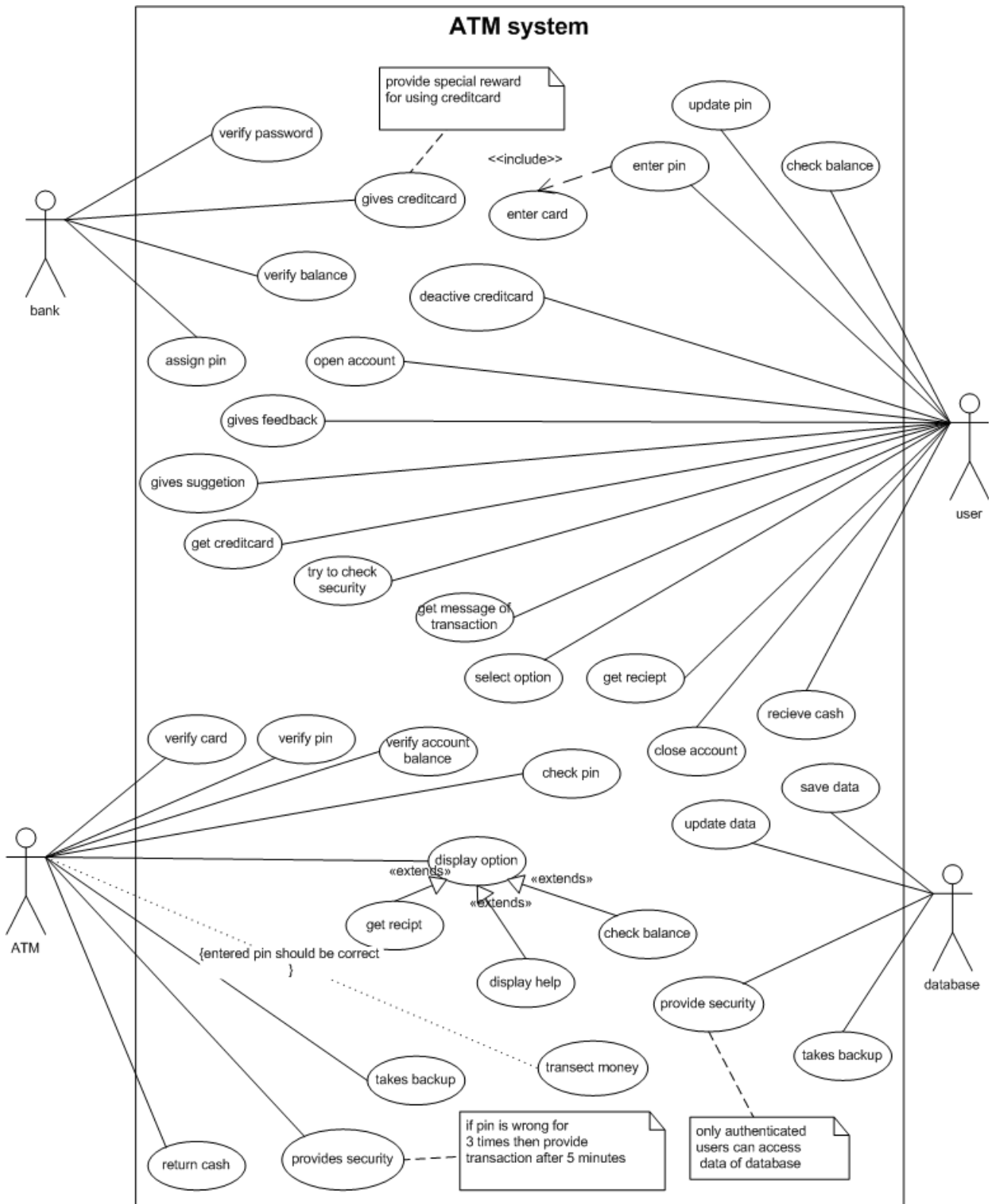
No.	Name	Notation	Description
1	System boundary		<p>The scope of a system can be represented by a system boundary. The use cases of the system are placed inside the system boundary, while the actors who interact with the system are put outside the system.</p> <p>The use cases in the system make up the total requirements of the system.</p>
2	Use case		<p>A use case represents a user goal that can be achieved by accessing the system or software application.</p>
3	Actor		<p>Actors are the entities that interact with a system. Although in most cases, actors are used to represent the users of system, actors can actually be anything that needs to exchange information with the system.</p> <p>So an actor may be people, computer hardware, other systems, etc. Note that actor represent a role that a user can play, but not a specific user.</p>

4	Association		Actor and use case can be associated to indicate that the actor participates in that use case. Therefore, an association corresponds to a sequence of actions between the actor and use case in achieving the use case.
5	Generalization		A generalization relationship is used to represent inheritance relationship between model elements of same type.
6	Include		An include relationship specifies how the behavior for the inclusion use case is inserted into the behavior defined for the base use case.
7	Extends		An extend relationship specifies how the behavior of the extension use case can be inserted into the behavior defined for the base use case.
8	Constraint		Show condition exists between actors an activity.
9	Package		Package is defined as collection of classes. Classes are unified together using a package.
10	Interface		Interface is used to connect package and use-case. Head is linked with package and tail linked with usecase.
11	Note		Note is generally used to write comment in use-case diagram.
12	Anchor		Anchor is used to connect a note the use case in use case diagram.

## Draw Use case diagram for Library management System

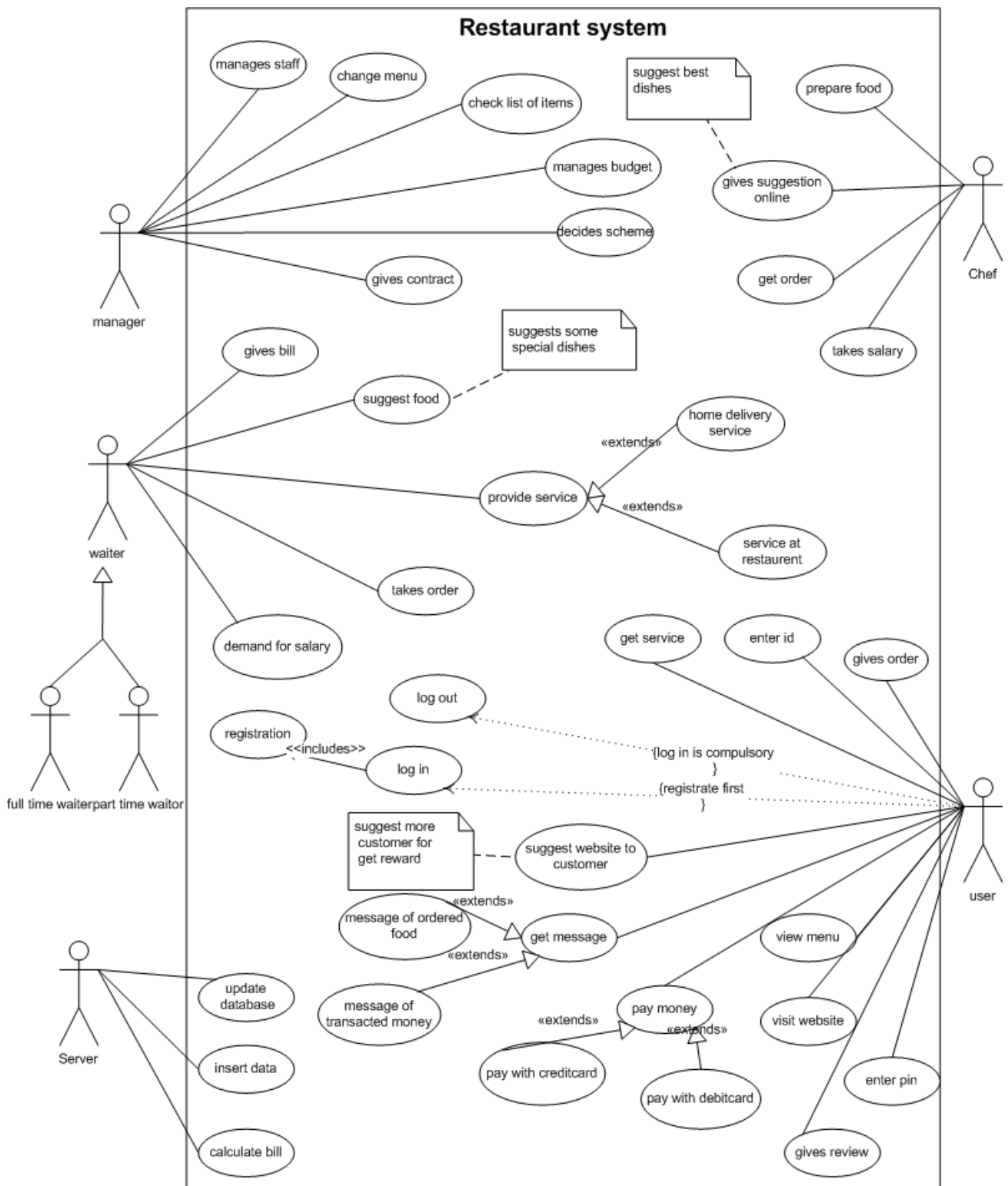


## Draw Use-case Diagram For ATM System



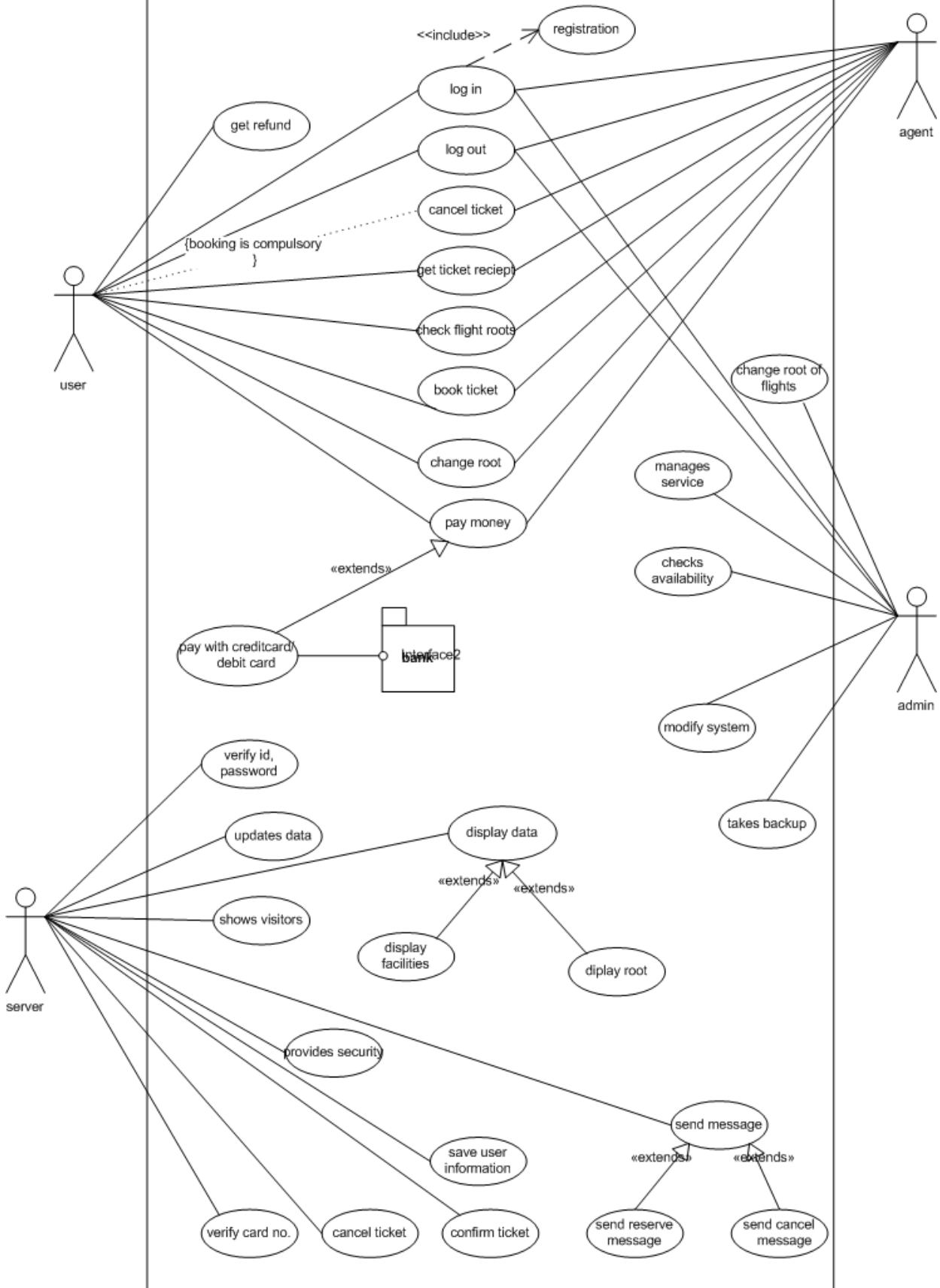


### Draw Use-case diagram for online restaurant system



# Draw Use-case for Online Reservation System

## ONLINE AIRWAY RESERVATION SYSTEM



## Draw Use-case diagram for online shopping system



## Sequence models

### Introduction

- Sequence diagrams model the dynamic aspects of a software system.
- The emphasis is on the “sequence” of messages rather than relationship between objects.
- A sequence diagram maps the flow of logic or flow of control within a usage scenario into a visual diagram enabling the software architect to both document and validate the logic during the analysis and design stages.
- Sequence diagrams provide more detail and show the message exchanged among a set of objects over time.
- Sequence diagrams are good for showing the behavior sequences seen by users of a diagram shows only the sequence of messages not their exact timing.
- Sequence diagrams can show concurrent signals.


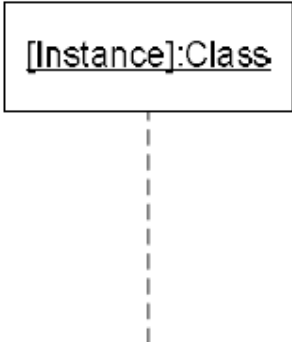
### Purpose

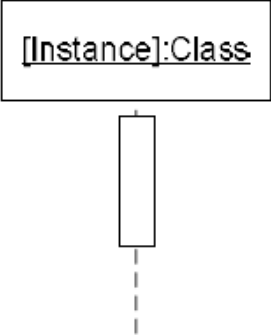
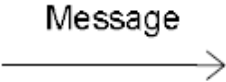
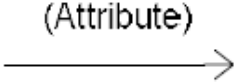
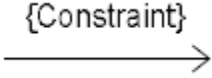
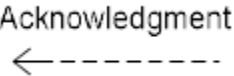
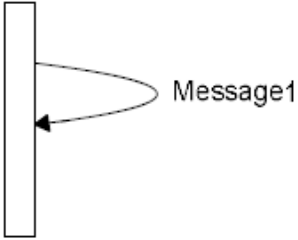

- The main purpose of this diagram is to represent how different business objects interact.
- A sequence diagram shows object interactions arranged in time sequence.
- It depicts the objects and classes involved in the scenario and the sequence of messages exchanged between the objects needed to carry out the functionality of the scenario.

### When to use : Sequence Diagram

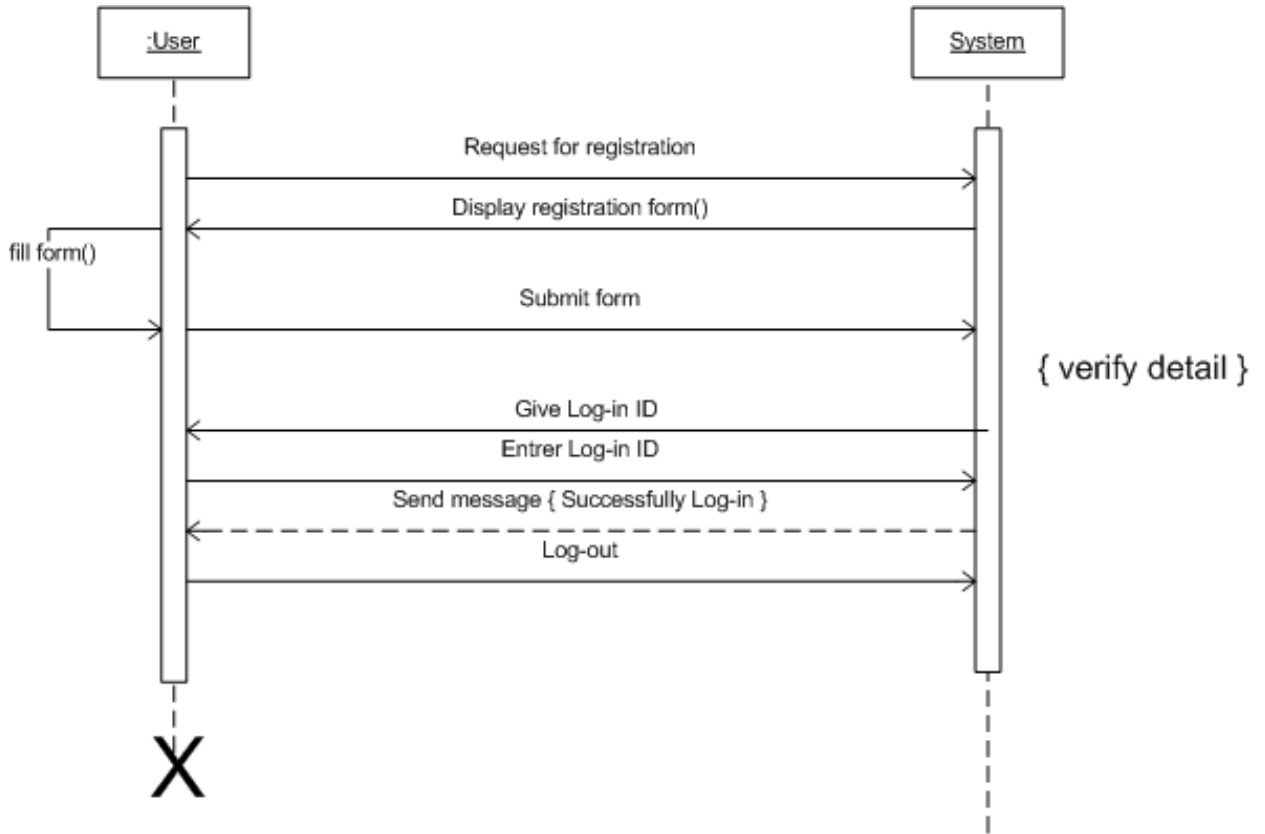
- Sequence diagram can be a helpful modeling tool when the dynamic behavior of objects needs to be observed in a particular use case or when there is a need for visualizing the “big picture of message flow”. A company’s technical staff could utilize sequence diagrams in order to document the behavior of a future system.
- It is during the design period that developers and architects utilize the diagram to showcase the system’s object interactions, thereby putting out a more fleshed out overall system design.

### Sequence Diagram Notations

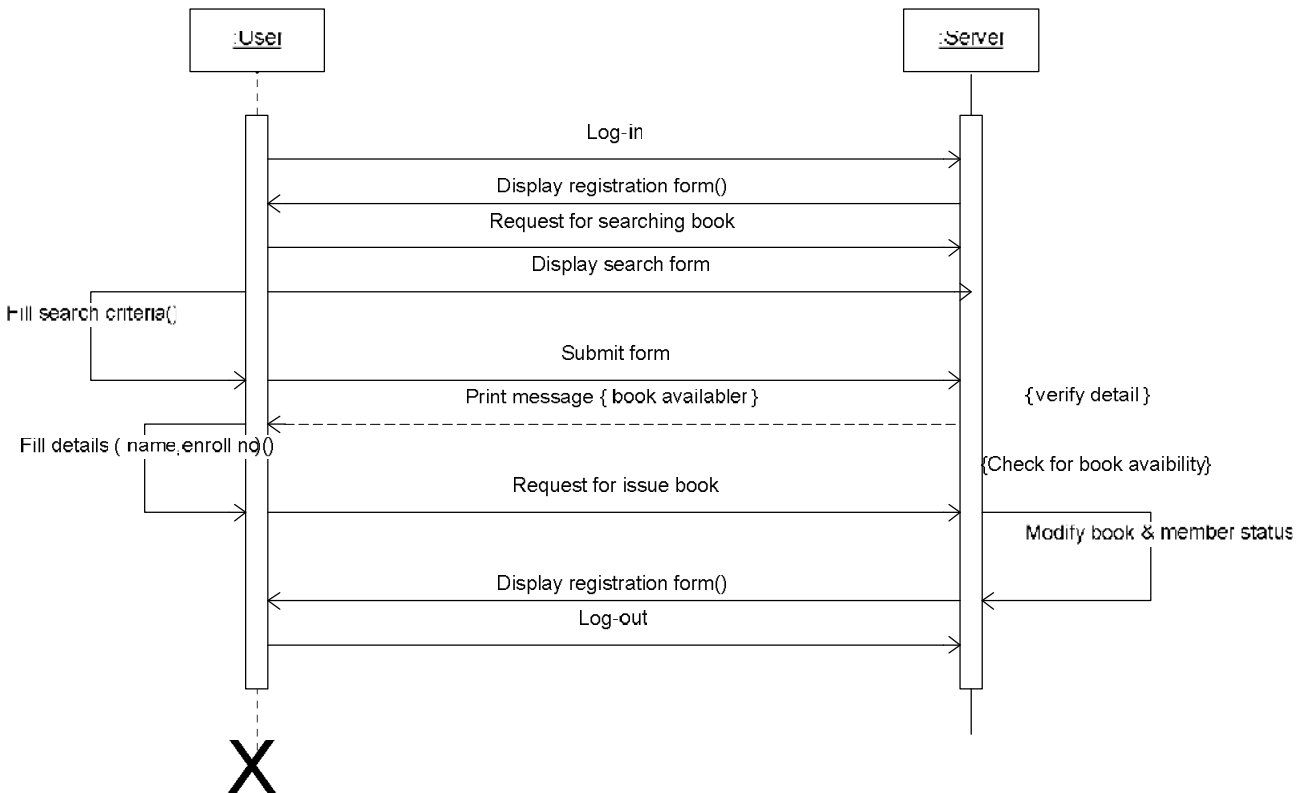
Sr. No.	Name	Notation	Description
1	Object		It represents the existence of an object of a particular time.
2	Life line		Lifeline represents the duration during which an object is alive and interacting with other objects in the system. It is represented by dashed lines.

3	Scope		It shows the time period during which an object or actor is performing an action.
4	Message transition		To send message from one object to another.
5	Message with attribute		To send message with some particular Attribute
6	Message with constraint		To send message from one object to other by some constraint.
7	Acknowledgement		It represents communication between objects conveys acknowledgement.
8	Self message		Self message occurs when an object sends a message to itself.
9	Recursive message		Self message occurs when an object sends a message to itself within recursive scope.

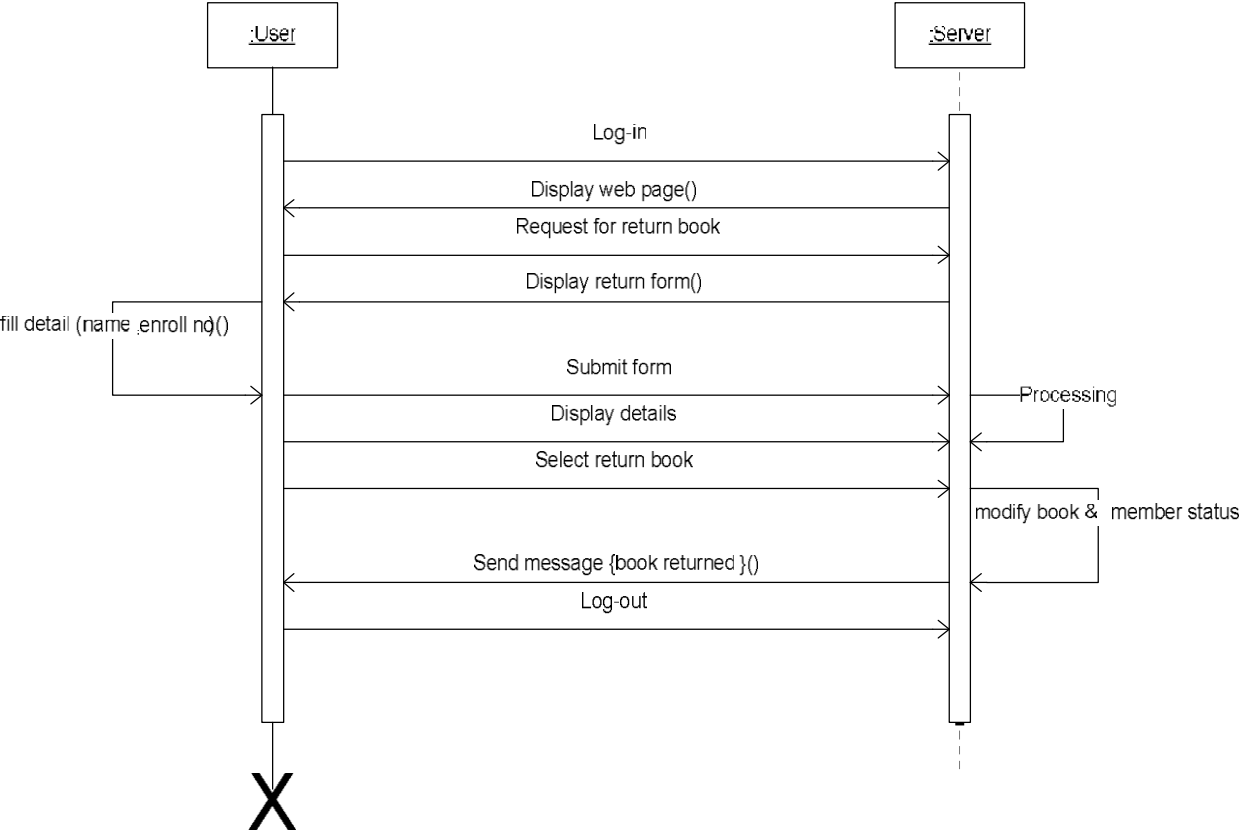
## Sequence Diagram for library management system



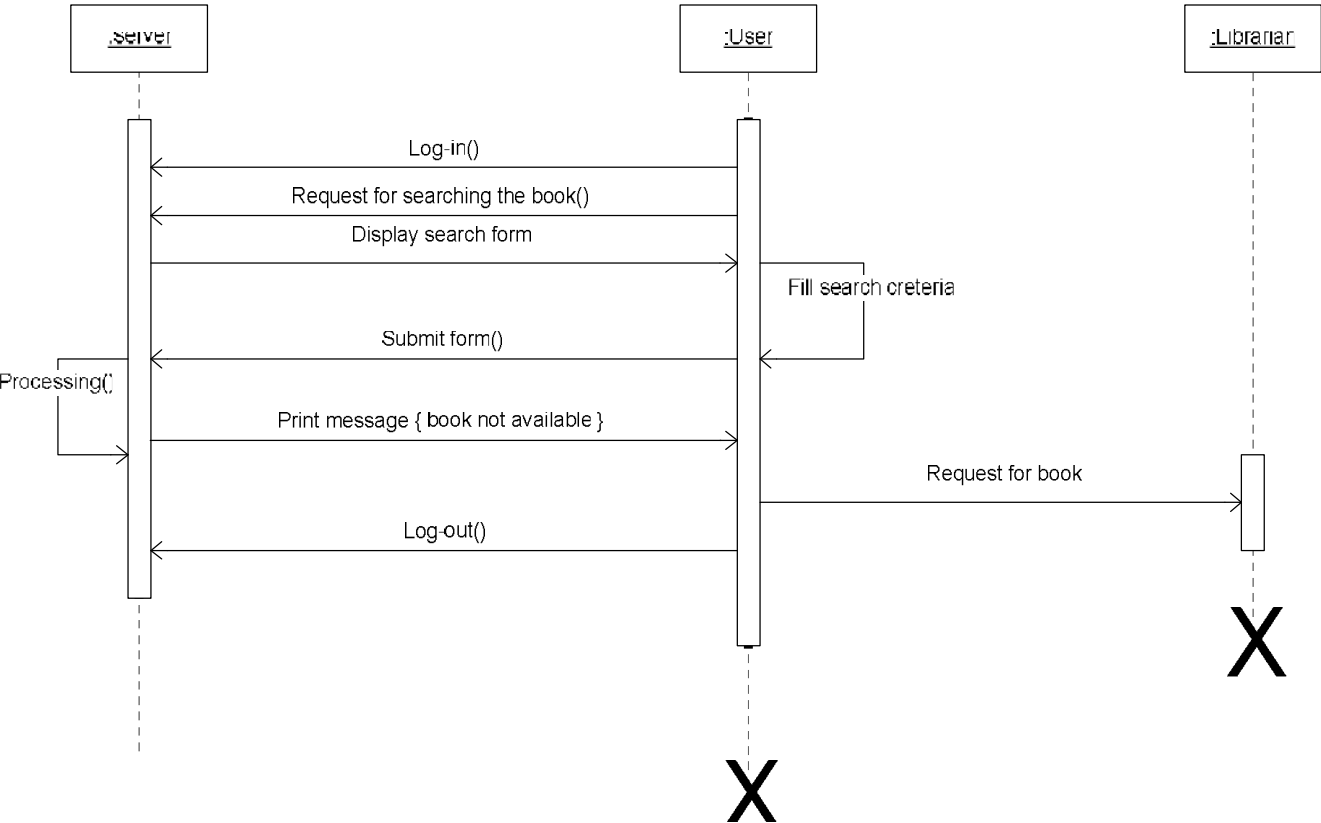
## Issue book



Return book

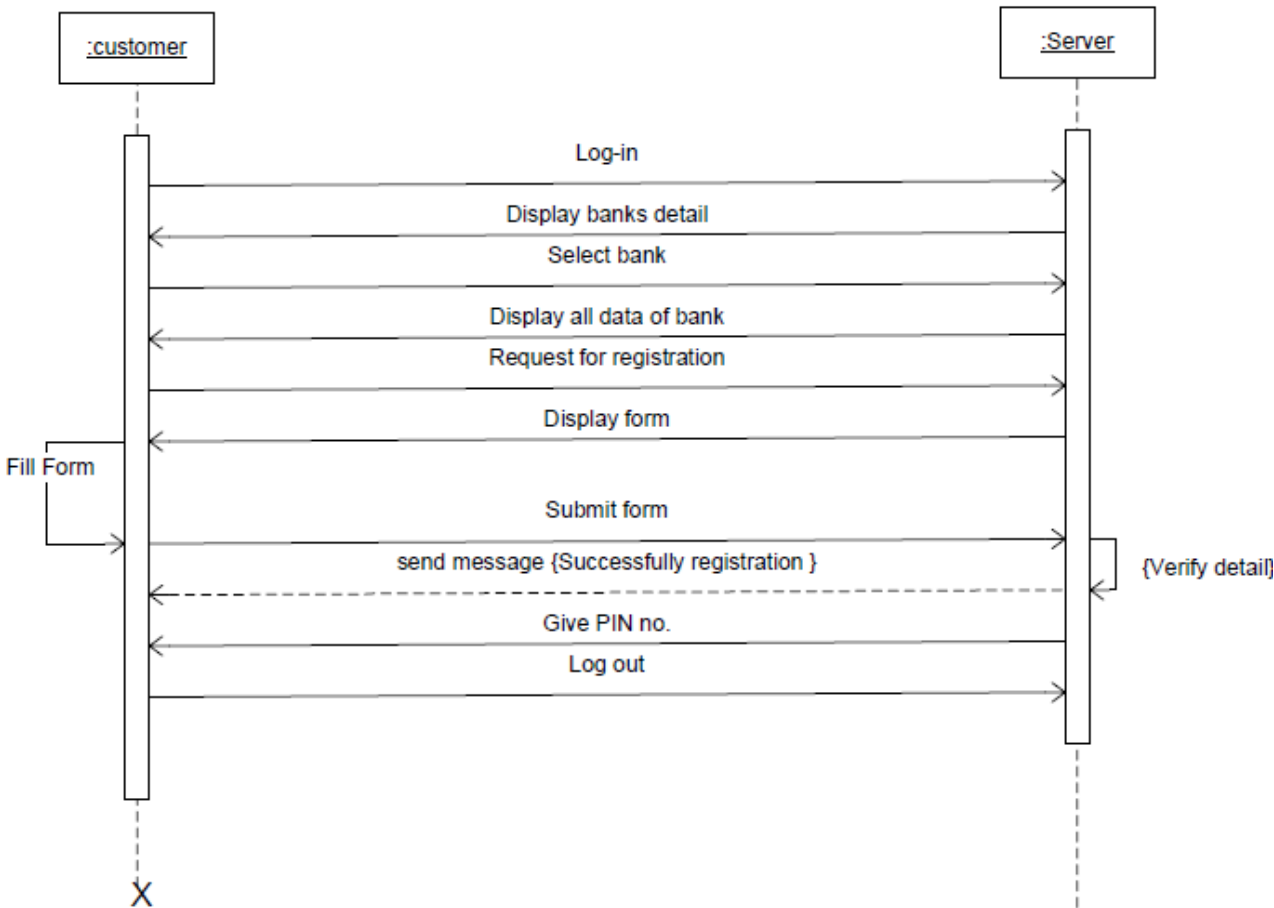


Book not available



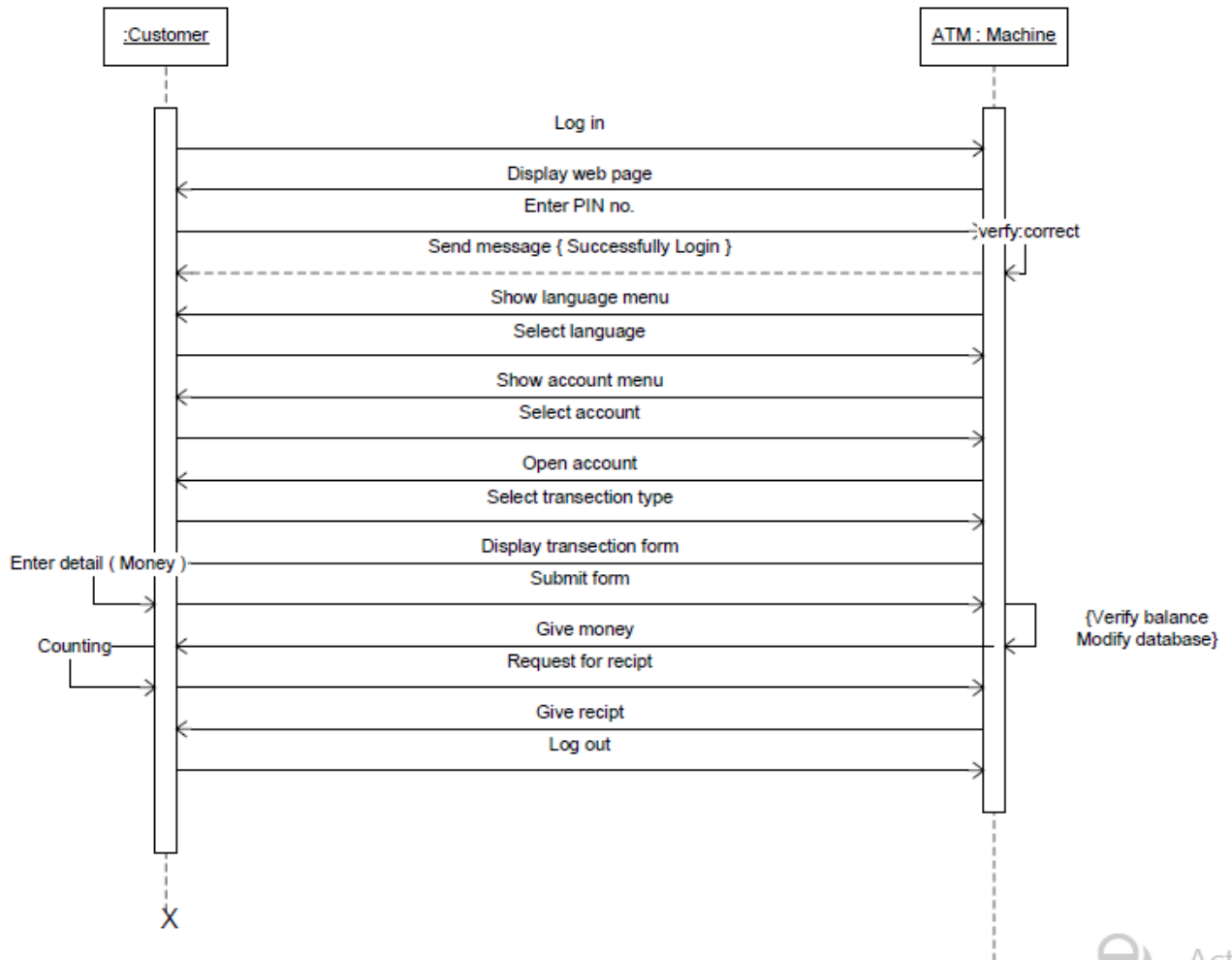
Sequence Diagram For ATM Management System:-

Create account

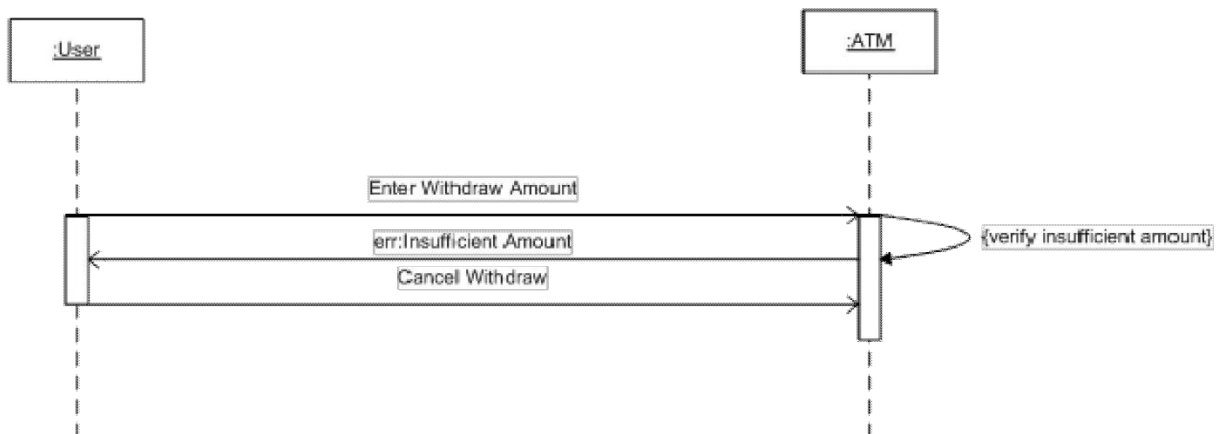




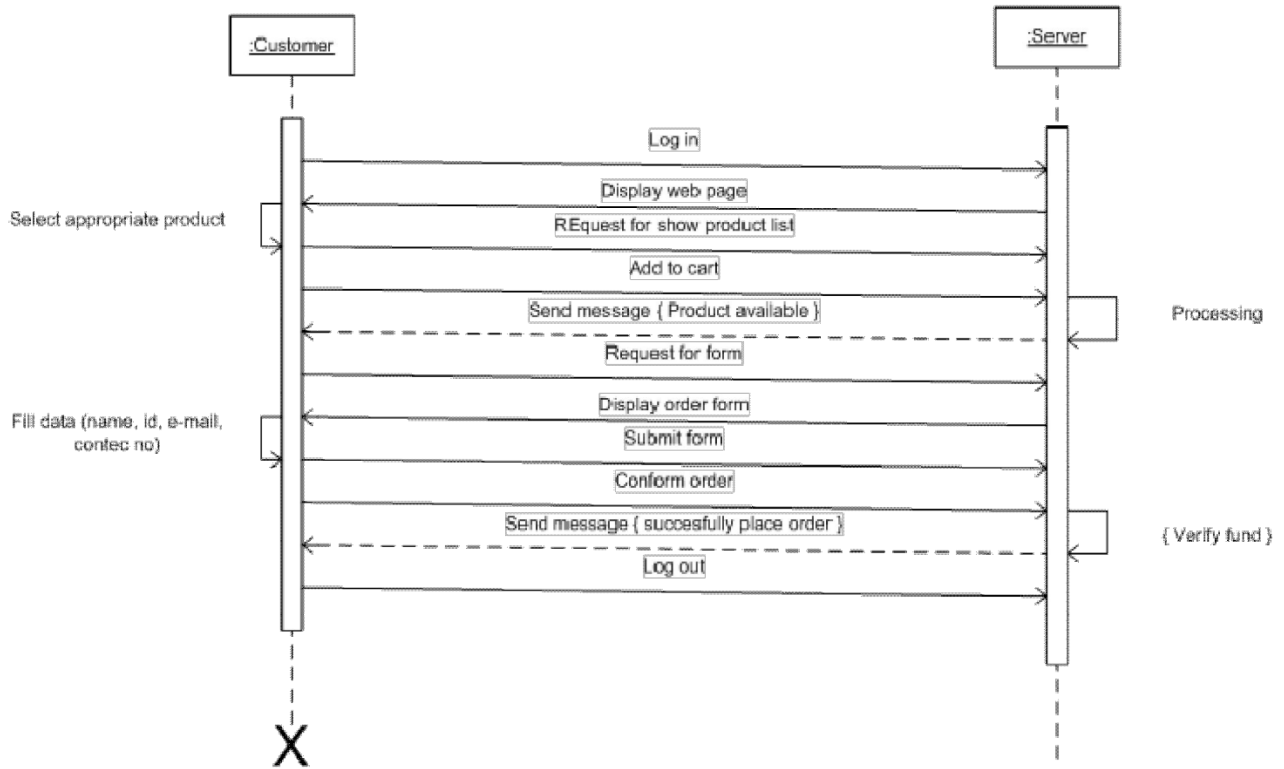
## Transaction



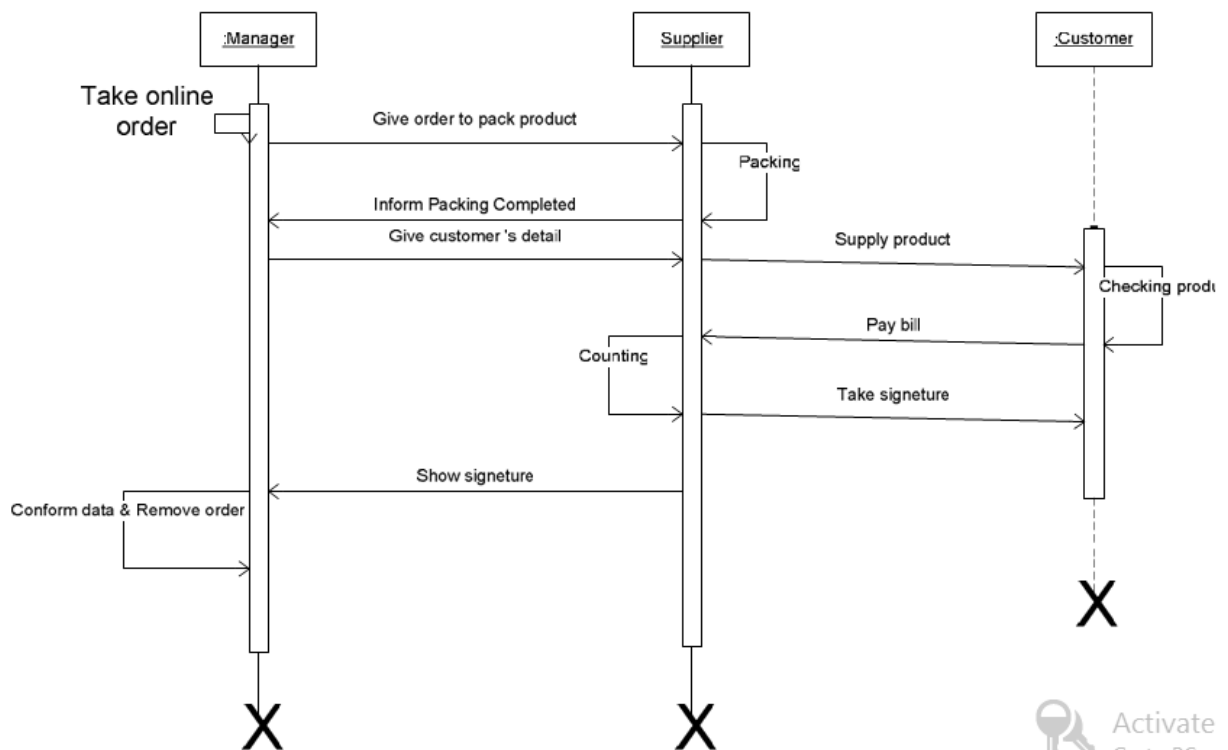
## Exceptional case



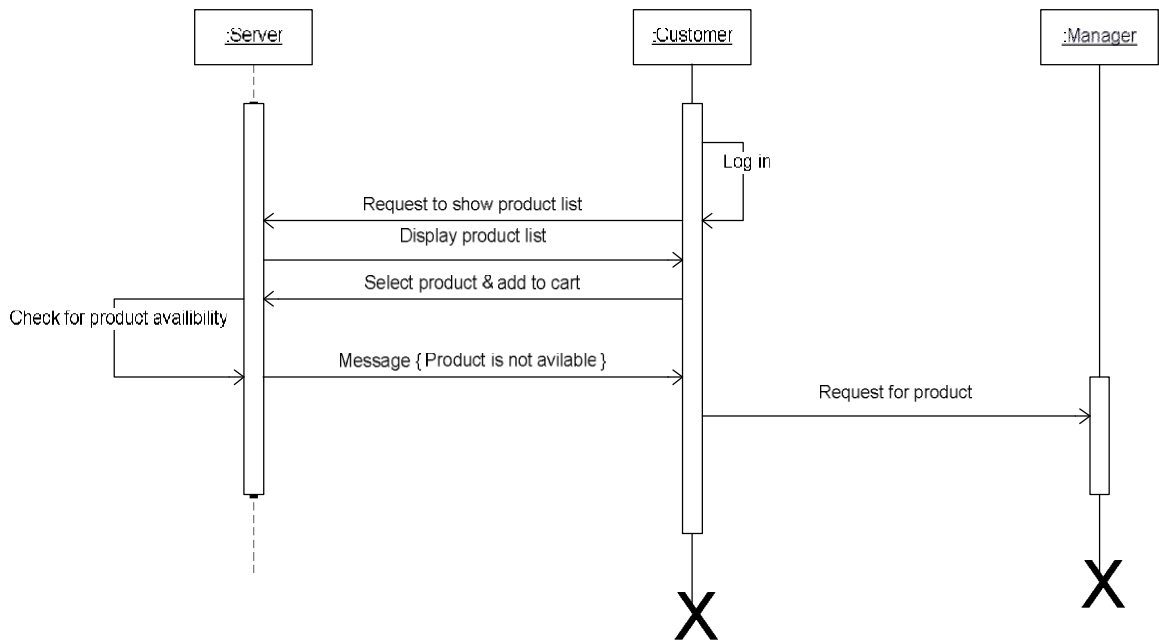
## Sequence diagram for Online shopping system to place order



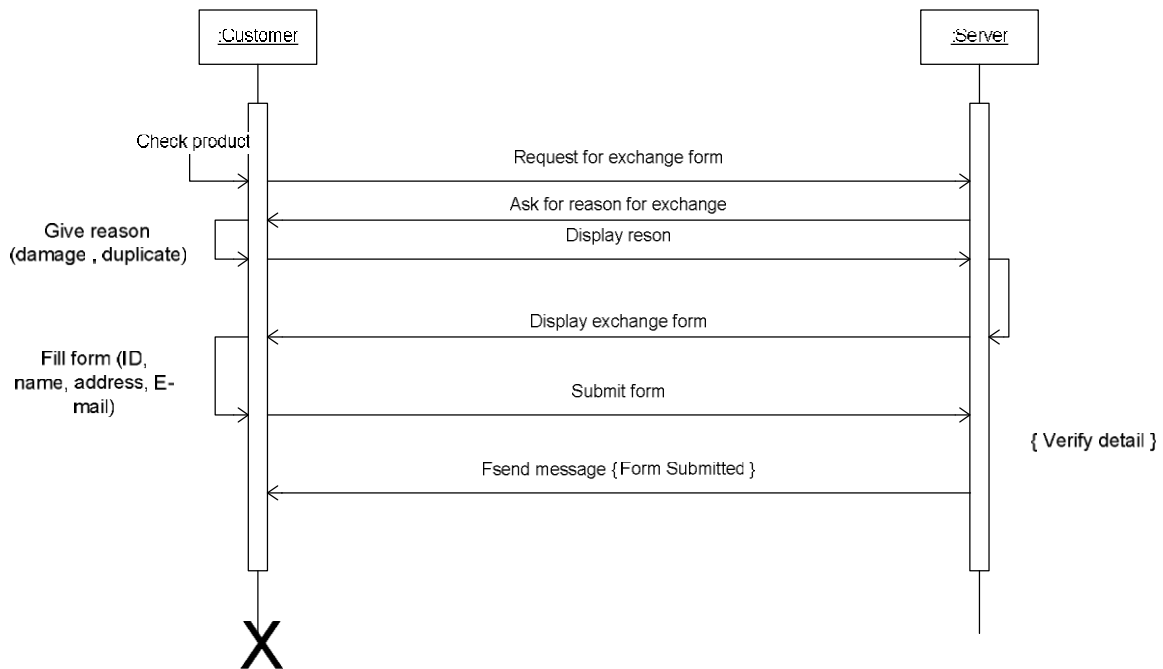
## Supply order



## Product not available

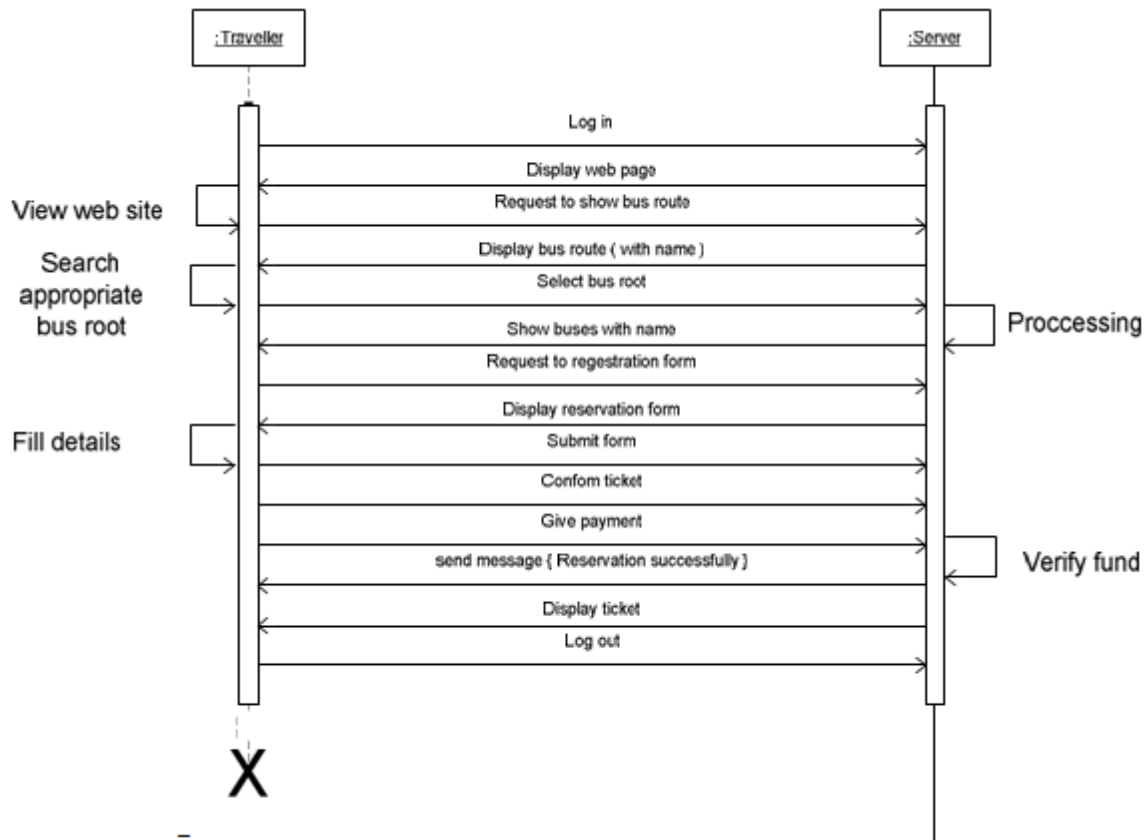


## Product Exchange

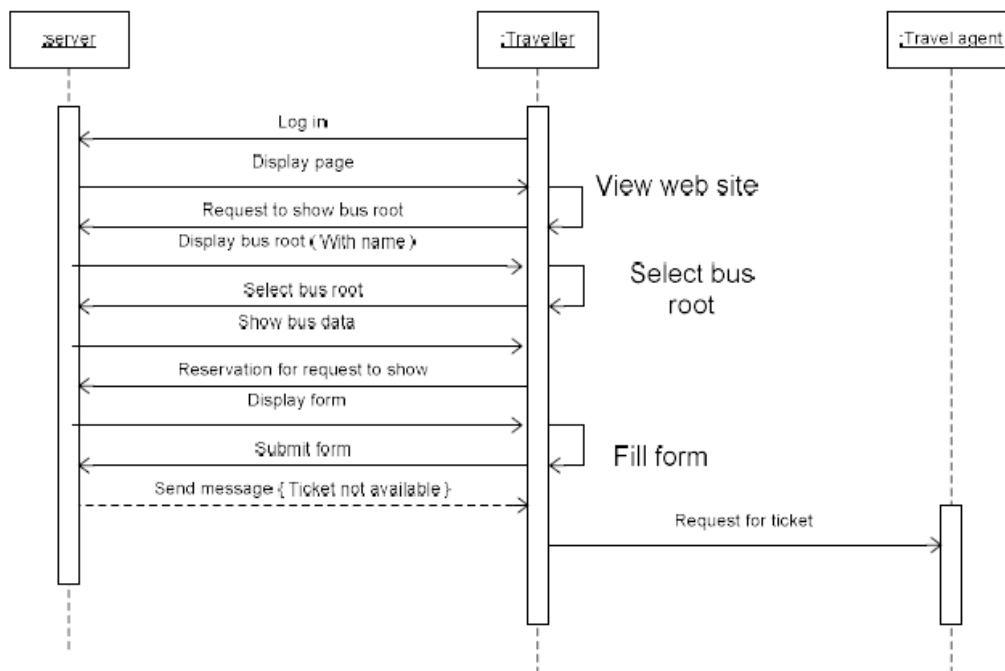


## Sequence diagram for Bus reservation system:-

### Reservation



### Ticket Not available



# Activity Diagram

## Introduction

- An activity diagram is a type of flow chart with additional support for parallel behavior.
- This diagram explains overall flow of control.
- Activity diagram is another important diagram in UML to describe dynamic aspects of the system.
- Activity diagram is basically a flow chart to represent the flow from one activity to another activity
- The activity can be described as an operation of the system.
- The control flow is drawn from one operation to another. This flow can be sequential, branched or concurrent. This distinction is important for a distributed system.
- Activity diagrams deals with all type of flow control by using different elements like fork, join etc.


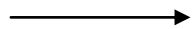
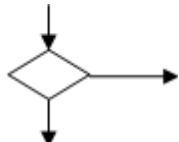
## Purpose



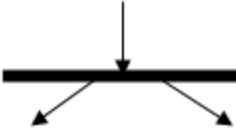
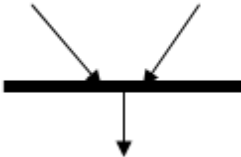
- Contrary to use case diagrams, in activity diagrams it is obvious whether actors can perform business usecases together or independently from one another.
- Activity diagrams allow you to think functionally.

## When to use : Activity Diagrams

- Activity diagrams are most useful when modeling the parallel behavior of a multithreaded system or when documenting the logic of a business process.
- Because it is possible to explicitly describe parallel events, the activity diagram is well suited for the illustration of business processes, since business processes rarely occur in a linear manner and often exhibit parallelisms.
- This diagram is useful to investigate business requirements at a later stage.
- An activity diagram is drawn from a very high level. So it gives high level view of a system. This high level view is mainly for business users or any other person who is not a technical person.
- This diagram is used to model the activities which are nothing but business requirements.
- So the diagram has more impact on business understanding rather implementation details.

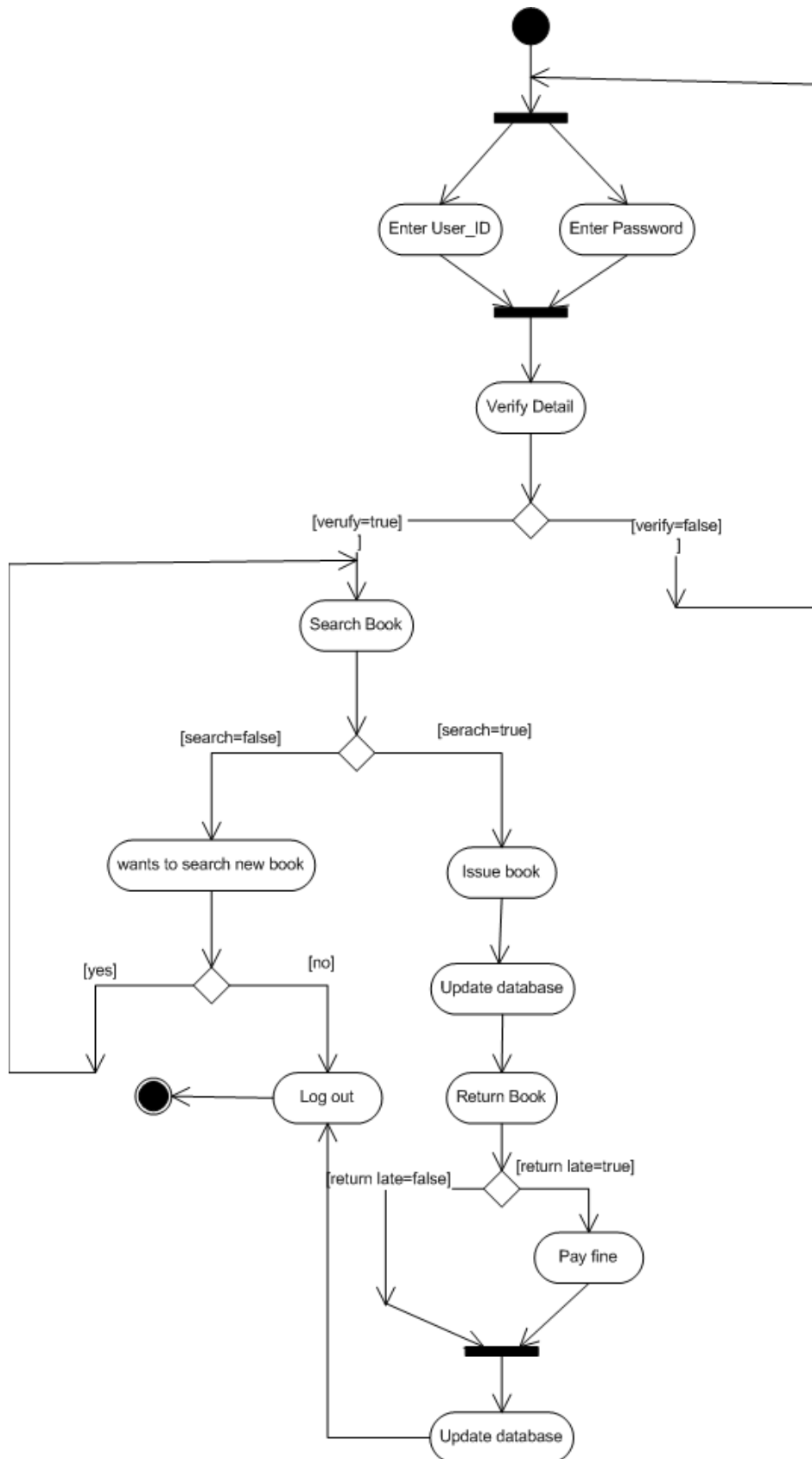
## Activity Diagram Notations

No.	Name	Symbol	Description
1	Activity		Represent individual activity of system.
2	Transition		Represents flow of data from one activity to another.
3	Decision		Decision node is a control node that accepts tokens on one or more incoming edges and selects outgoing edge from two or more outgoing flows. The notation for a decision node is a diamond-shaped symbol.

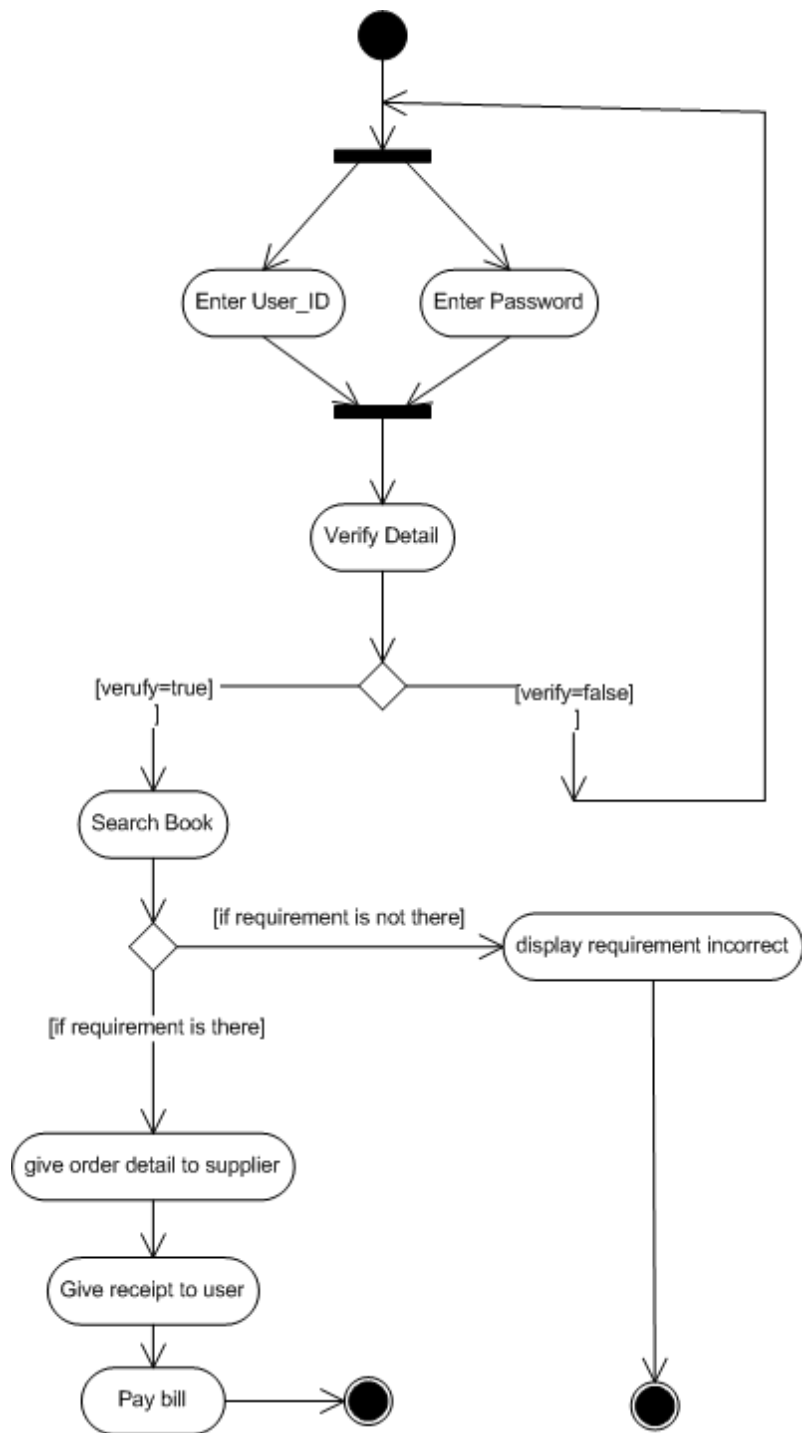
4	Initial activity		Initial node is a control node at which flow starts when the activity is invoked. Activity may have more than one initial node. Initial nodes are shown as a small solid circle.
5	Final activity		Final node is a control final node that stops all flows in an activity. Activity final nodes are shown as a solid circle with a hollow circle inside. It can be thought of as a goal notated as "bull's eye," or target.
6	Fork		A fork in the activity diagram has a single incoming transition and multiple outgoing transitions exhibiting parallel behavior. The incoming transition triggers the parallel outgoing transitions.
7	Join		A join in the activity diagram synchronizes the parallel behavior started at a fork. Join ascertains that all the parallel sets of activities (irrespective of the order) are completed before the next activity starts. It is a synchronization point in the diagram. Each fork in an activity diagram has a corresponding join where the parallel behavior terminates.

## Activity Diagram for Library Management System

### Issue and return book



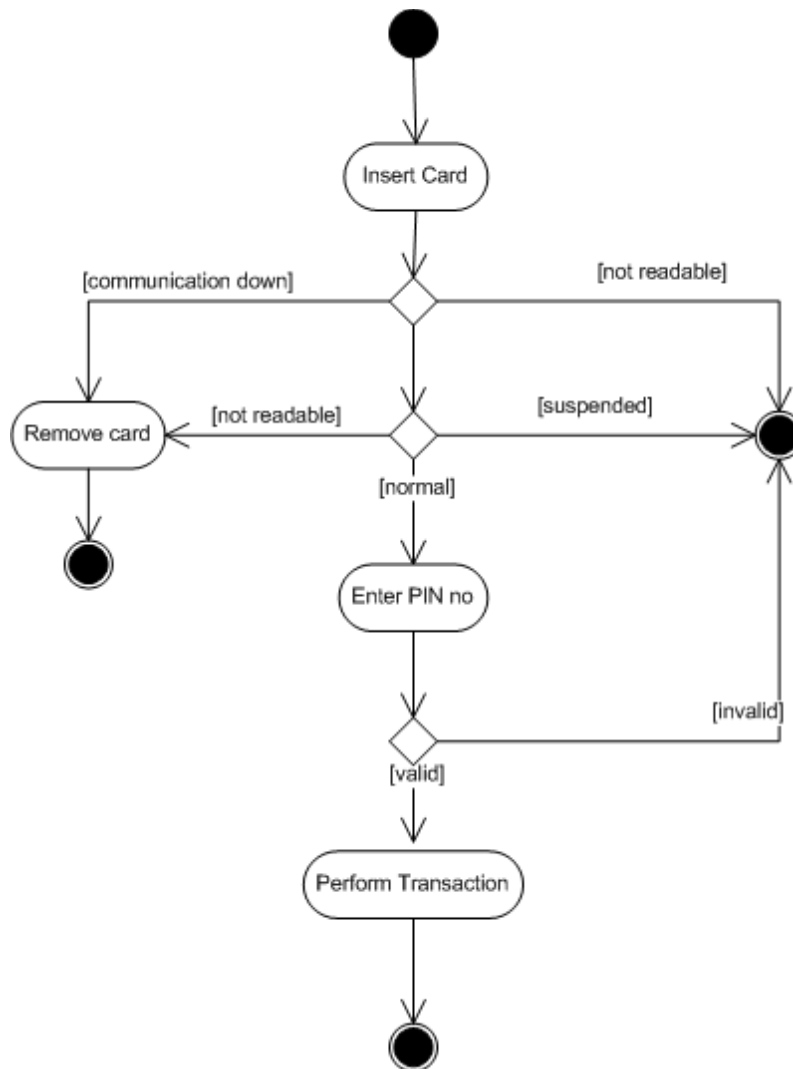
## Diagram for ordering new book



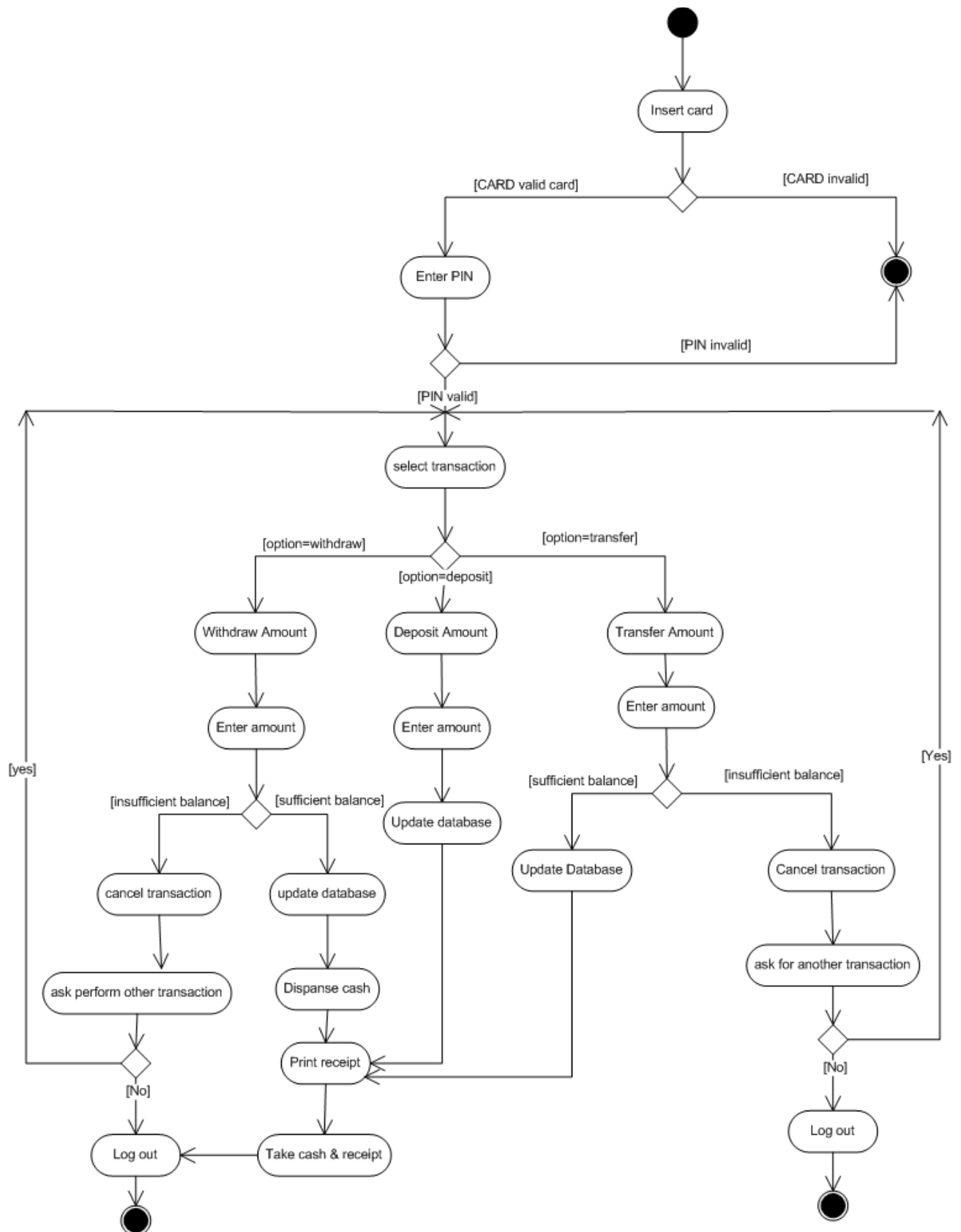


## Activity diagram for ATM

### Verify PIN number

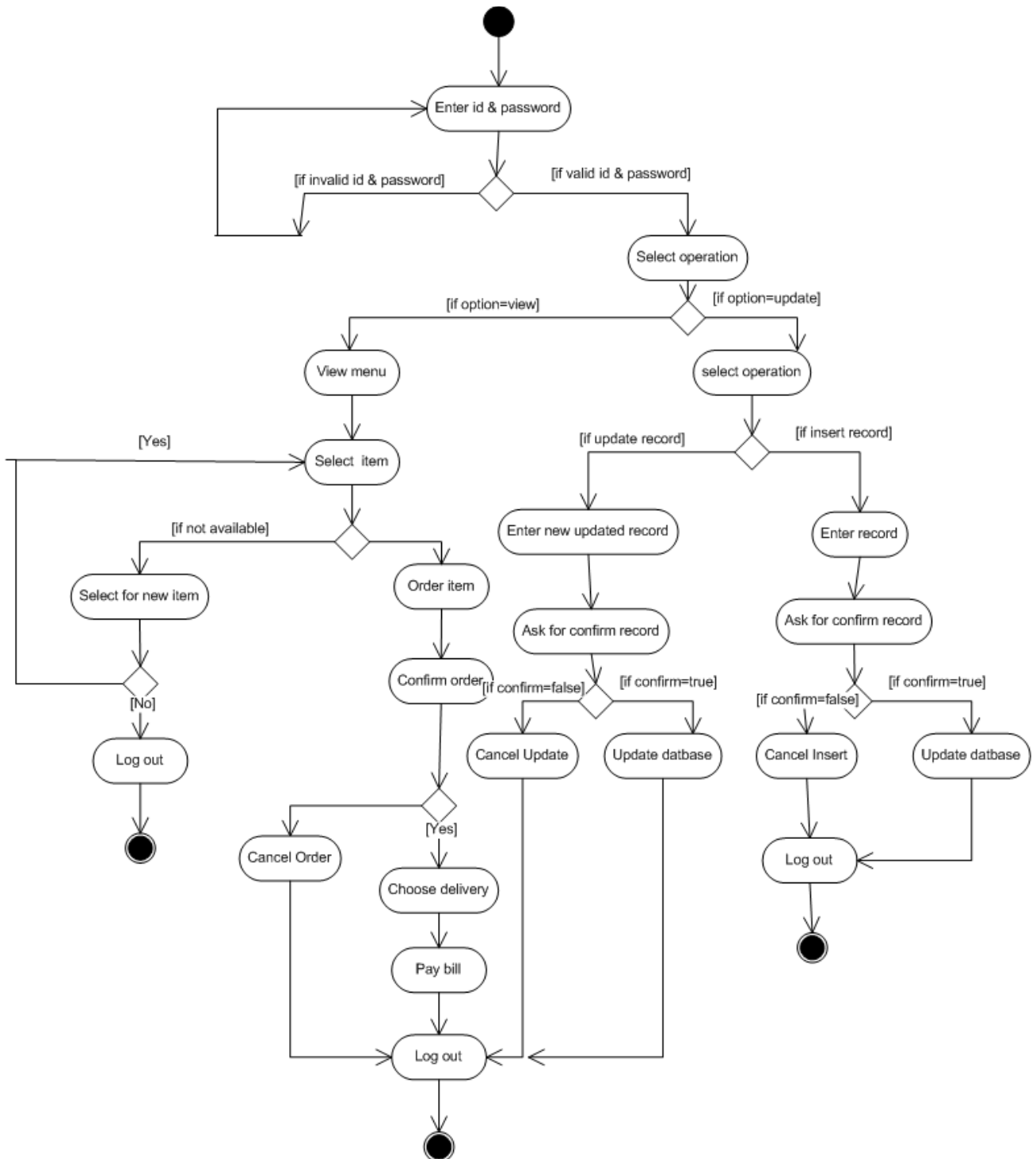


## Transaction

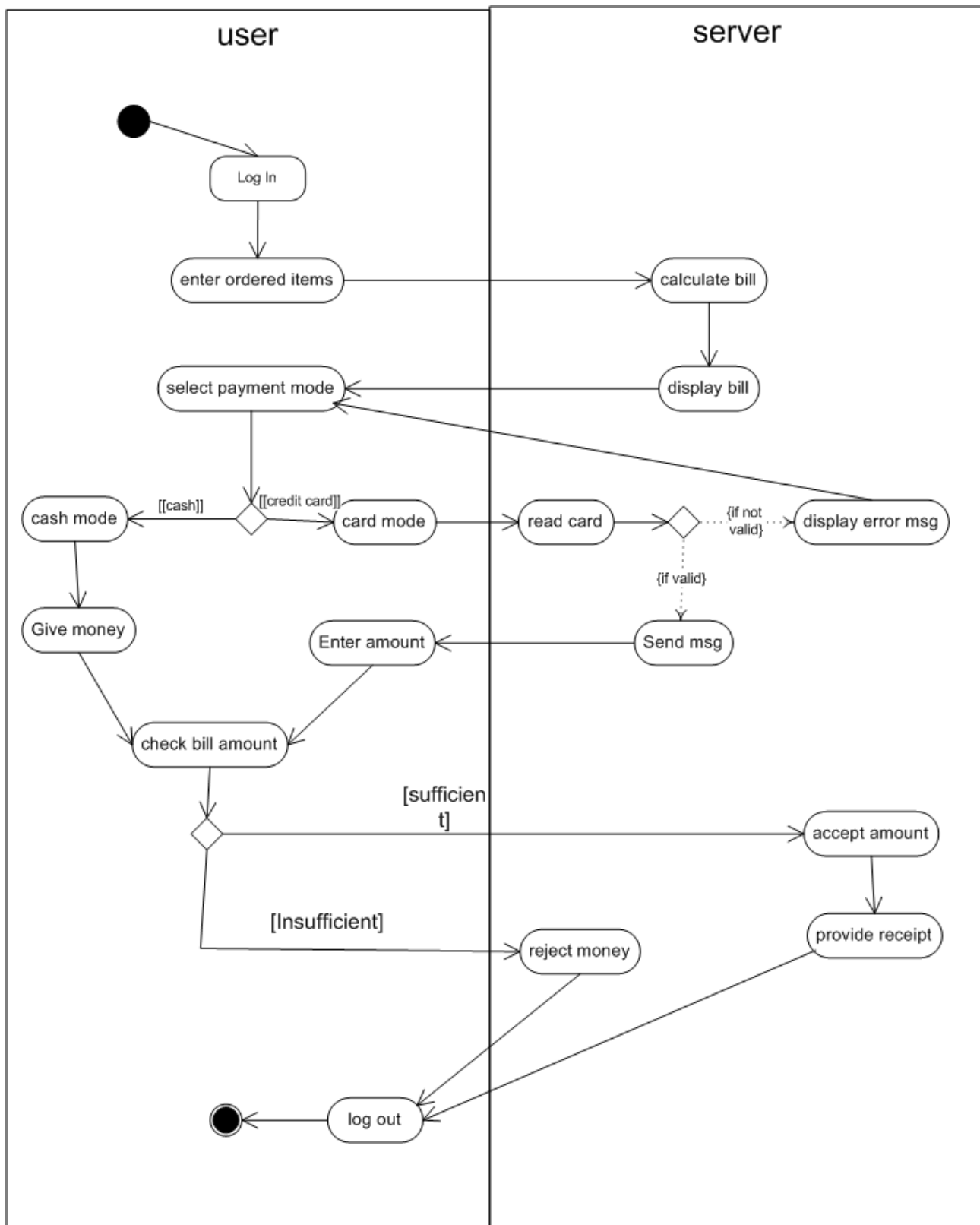


## Activity Diagram for Online Restaurant Management System

### Place order

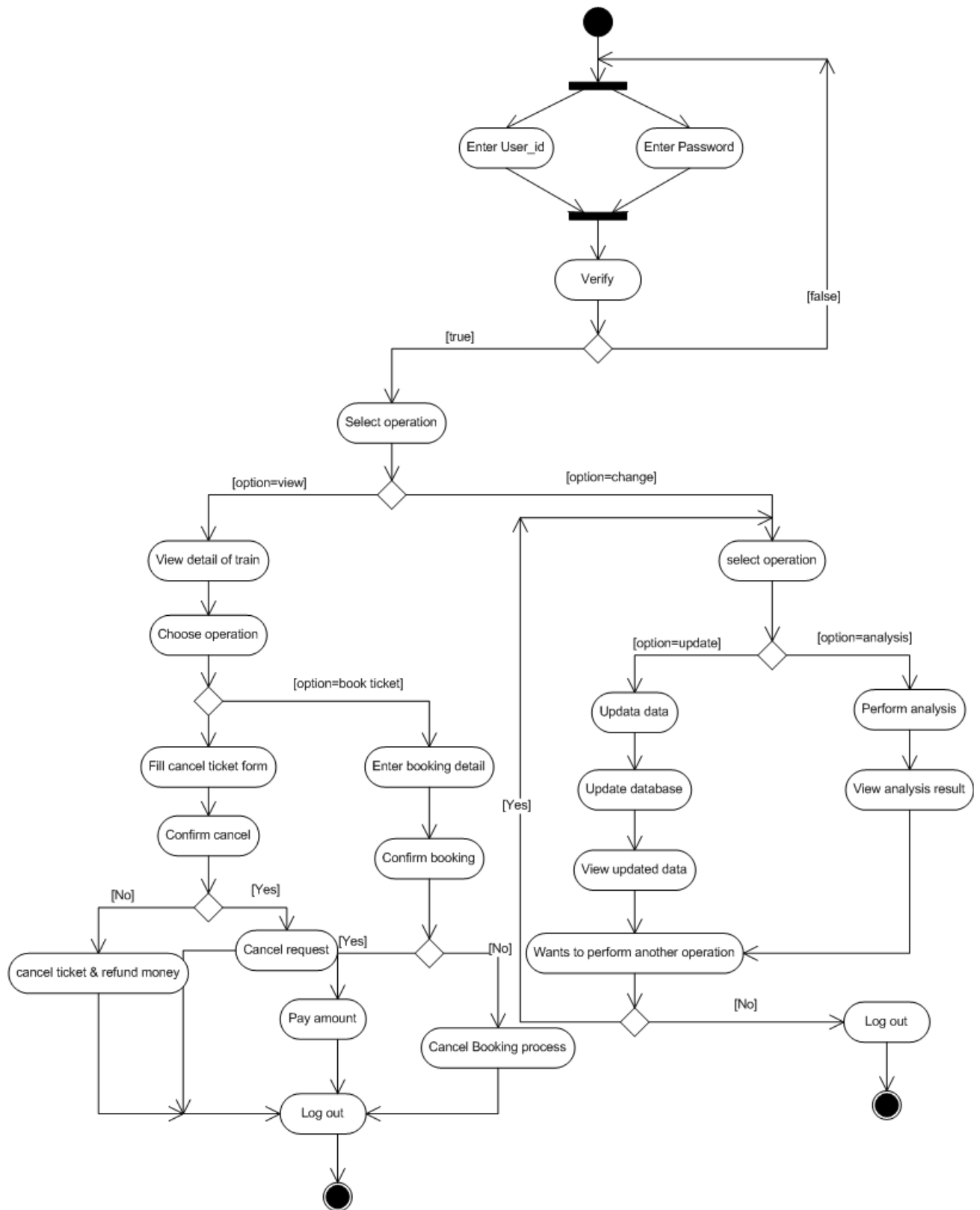


## Payment

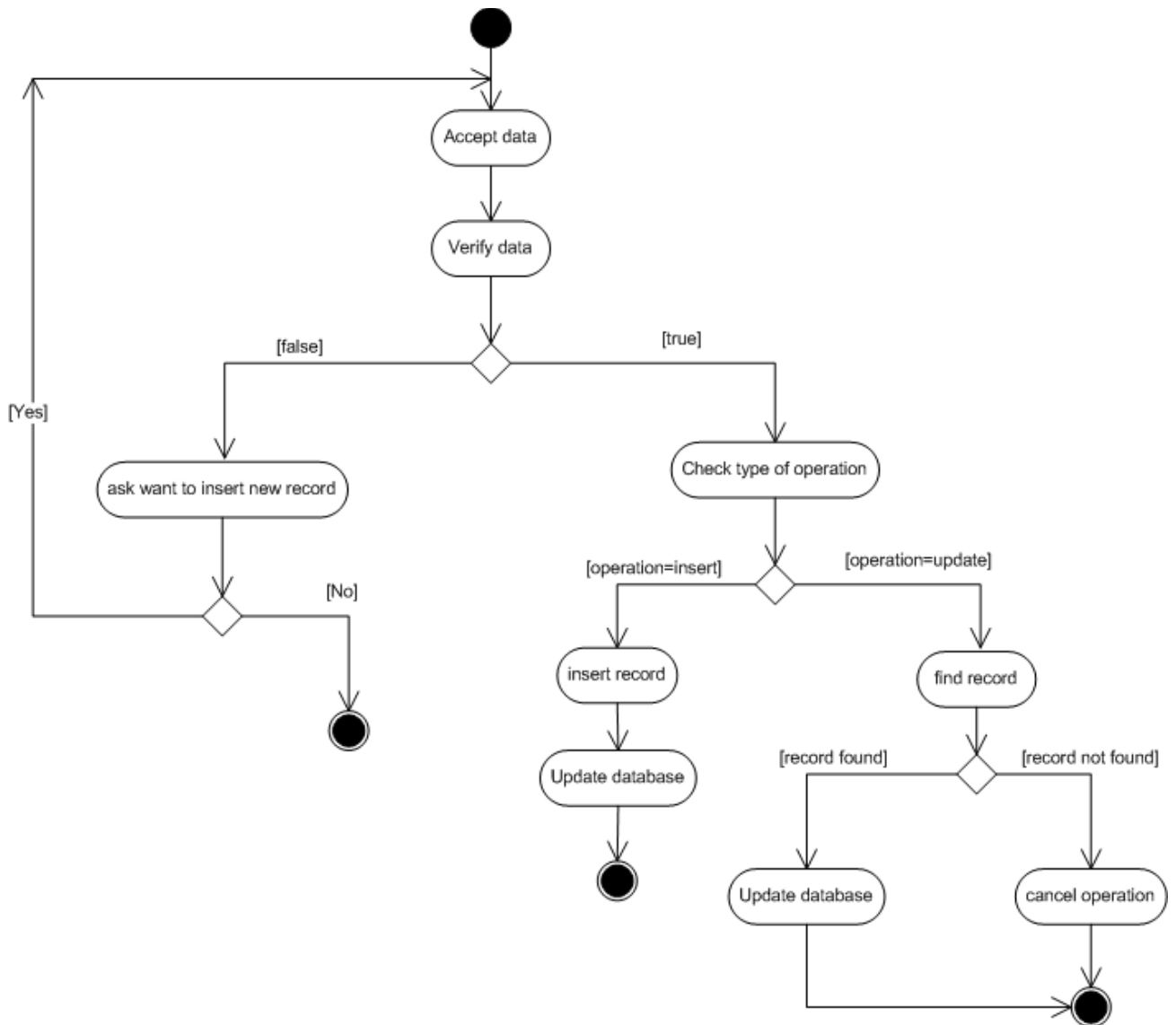


## Activity Diagram for Online Reservation System

### Booking Process



## Server Operation



## Activity diagram for Online Shopping

### Purchase Product

