

Software Design and Analysis
CS-3004
Lecture#02

Dr. Javaria Imtiaz,
Mr. Basharat Hussain,
Mr. Majid Hussain

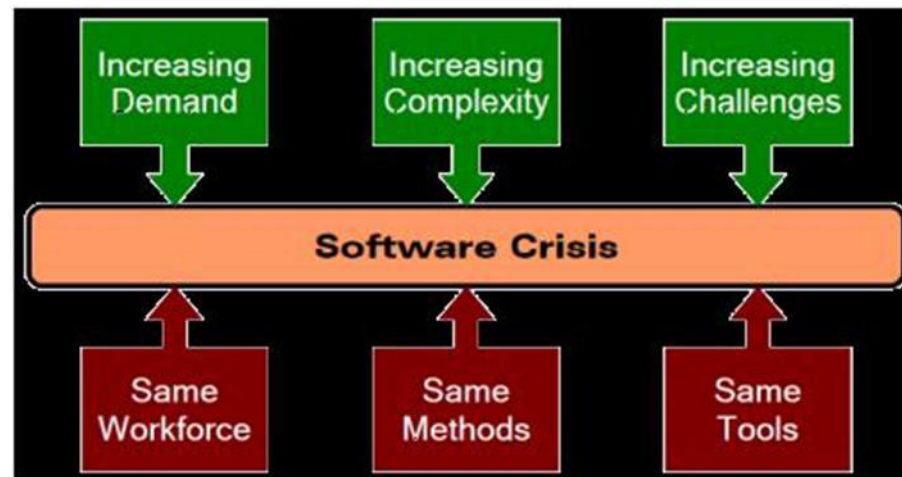
Agenda

- Traditional Software development methodologies
- Software Crises
- Types of Software Architectures
- Why Object Oriented Paradigm (OOP)?
- Object oriented Principles

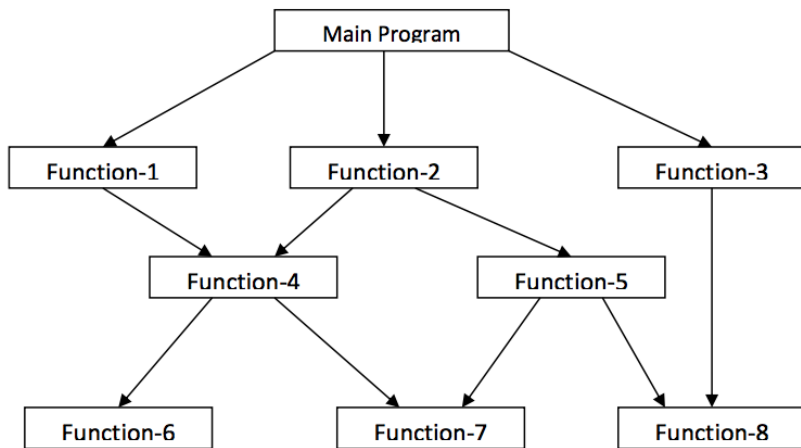
Goal of this Course

How to engineer software so that it meets customer's expectations & doesn't contribute to the **software crisis**

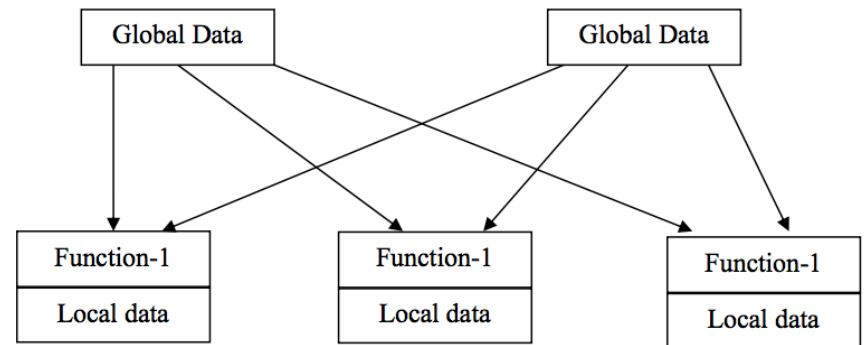
What is Software Crisis?



Traditional Software development



Code and Fix approach



Software Crisis – Observations

Software products:

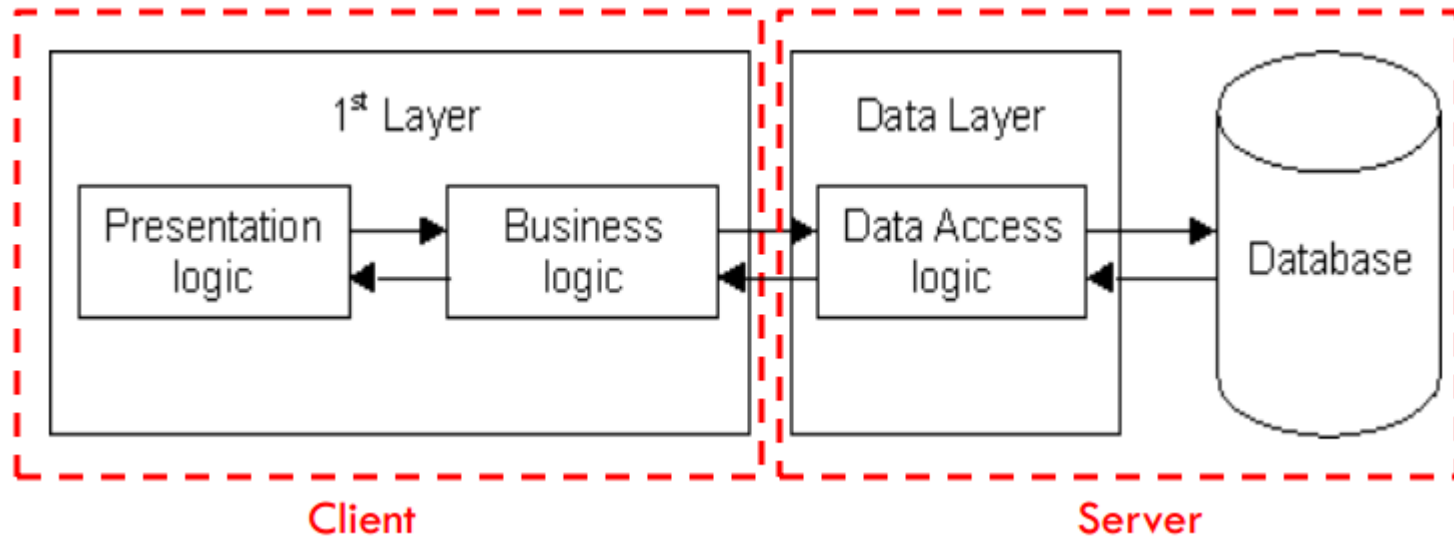
- fail to meet user requirements
- crash frequently
- expensive
- difficult to alter, debug, enhance
- often delivered late
- use resources non-optimally

Significance of “Tiers”

- **N-tier architectures have the same components**
 - Presentation
 - Business/Logic
 - Data
- **N-tier architectures try to separate the components into different tiers/layers**
 - Tier: physical separation
 - Layer: logical separation

Splitting software into several definite domains that handle particular aspects of the software such as presentation, logic or data management

2-Tier Architecture



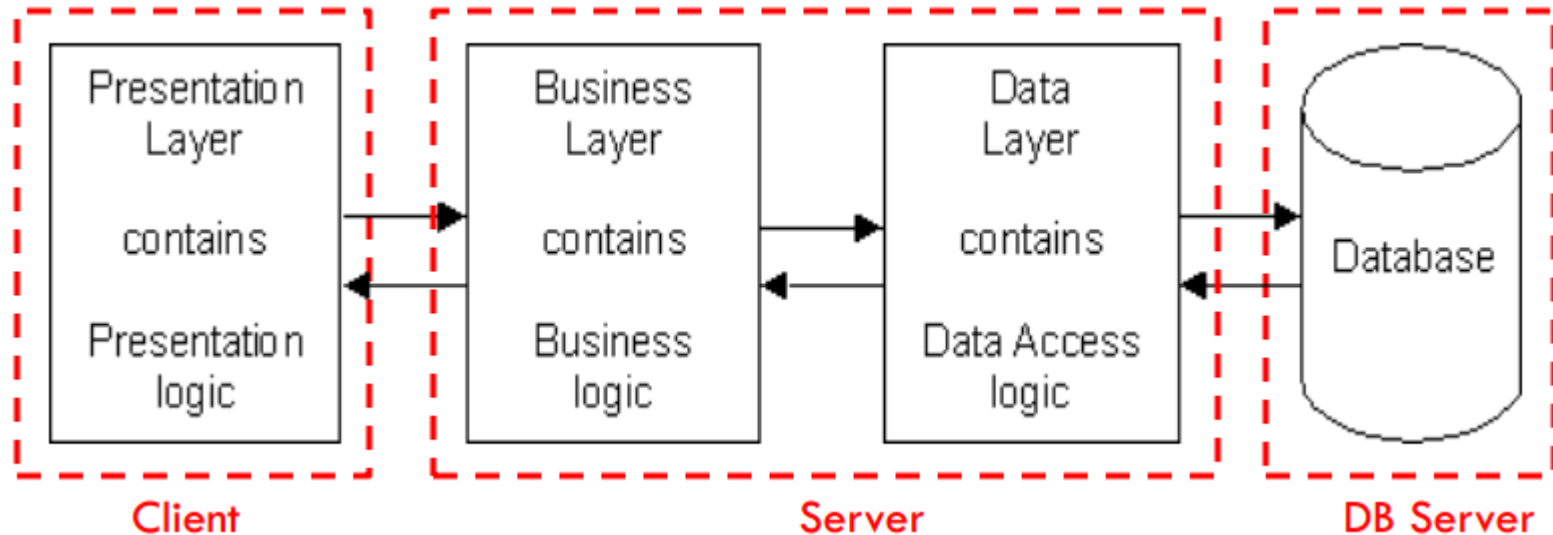
Database runs on Server

- Separated from client
- Easy to switch to a different database

Presentation and logic layers still tightly connected (coupled)

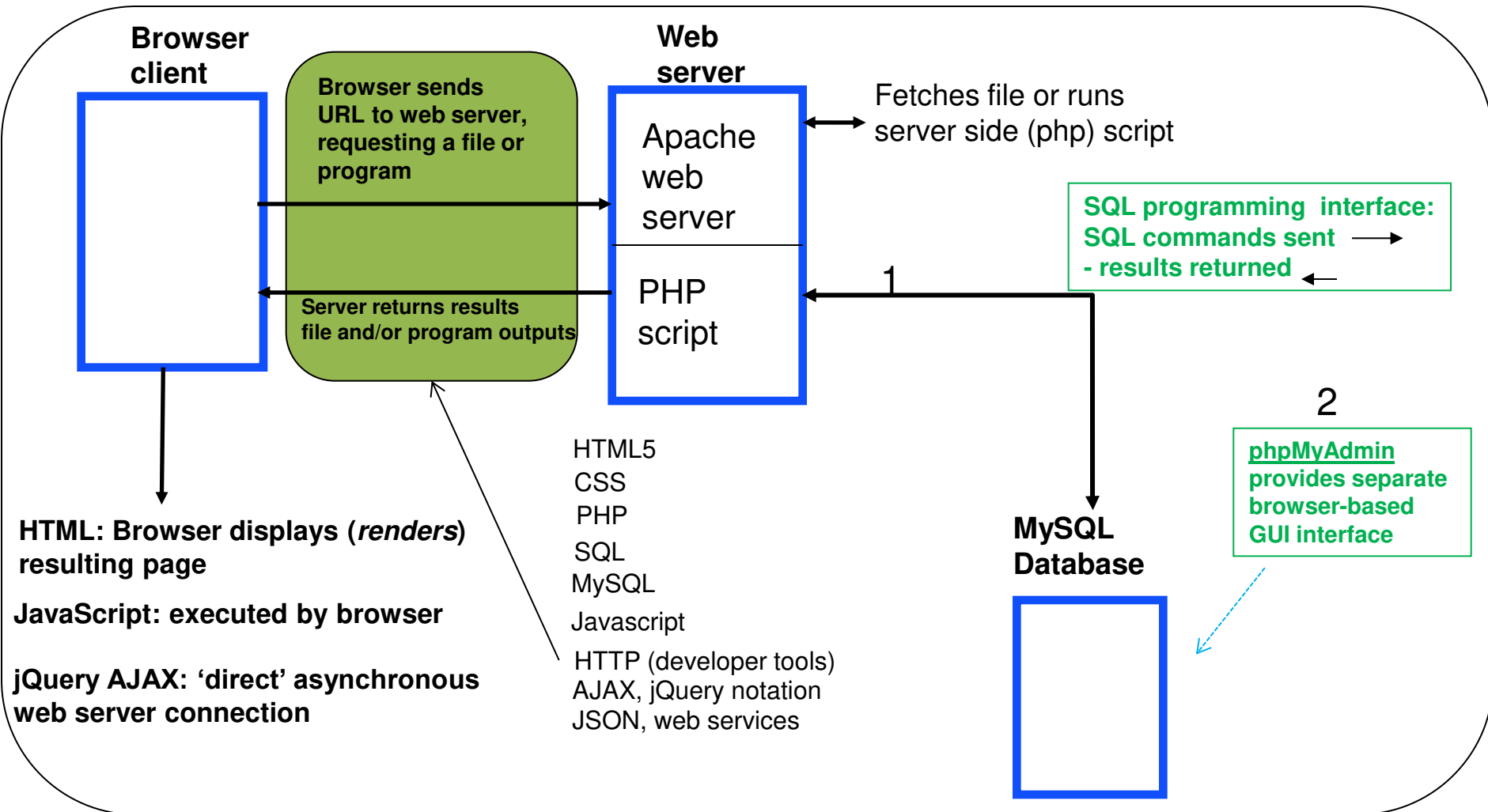
- Heavy load on server
- Potential congestion on network
- Presentation still tied to business logic

3-Tier Architecture



- Each layer can potentially run on a different machine
- Presentation, logic, data layers disconnected

Example of 3-Tier Architecture (web-based applications)

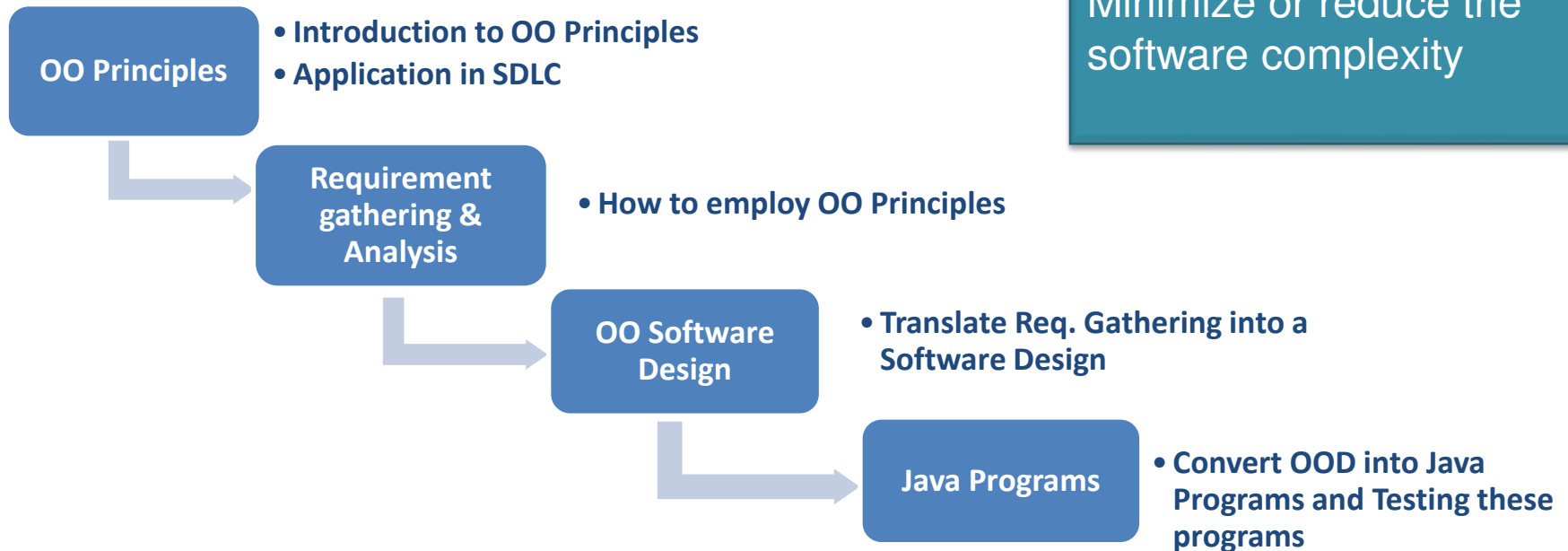


Course Objectives

What we will learn:

Techniques: Object Oriented Approach Using UML and Java

Minimize or reduce the software complexity



The Object-Oriented ... Hype

- **What are object-oriented (OO) methods?**
 - OO methods provide a set of techniques for analysing, decomposing, and modularising software system architectures
 - In general, OO methods are characterized by structuring the system architecture on the basis of its *objects* (and classes of objects) rather than the *actions* it performs
- **What is the rationale for using OO?**
 - In general, systems evolve and functionality changes, but objects and classes tend to remain stable over time
 - Use it for **large systems**
 - Use it for **systems that change often**

OO Design vs. OO Programming

- **Object-Oriented Design**

- a method for decomposing software architectures
- based on the objects every system or subsystem manipulates
- relatively independent of the programming language used

- **Object-Oriented Programming**

- construction of software systems as
 - Classes
 - Inheritance
 - Polymorphism
- concerned with programming languages and implementation issues

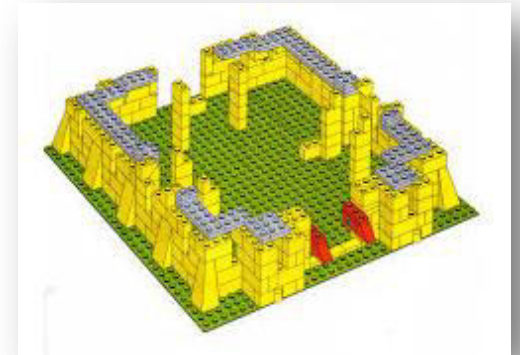
OO Paradigm– Analogy (Legos)



Each Lego part is a small object



That fits together with other small objects in pre-defined way



In order to create other larger objects



That is roughly how object-oriented programming works: putting together smaller elements to build larger ones.

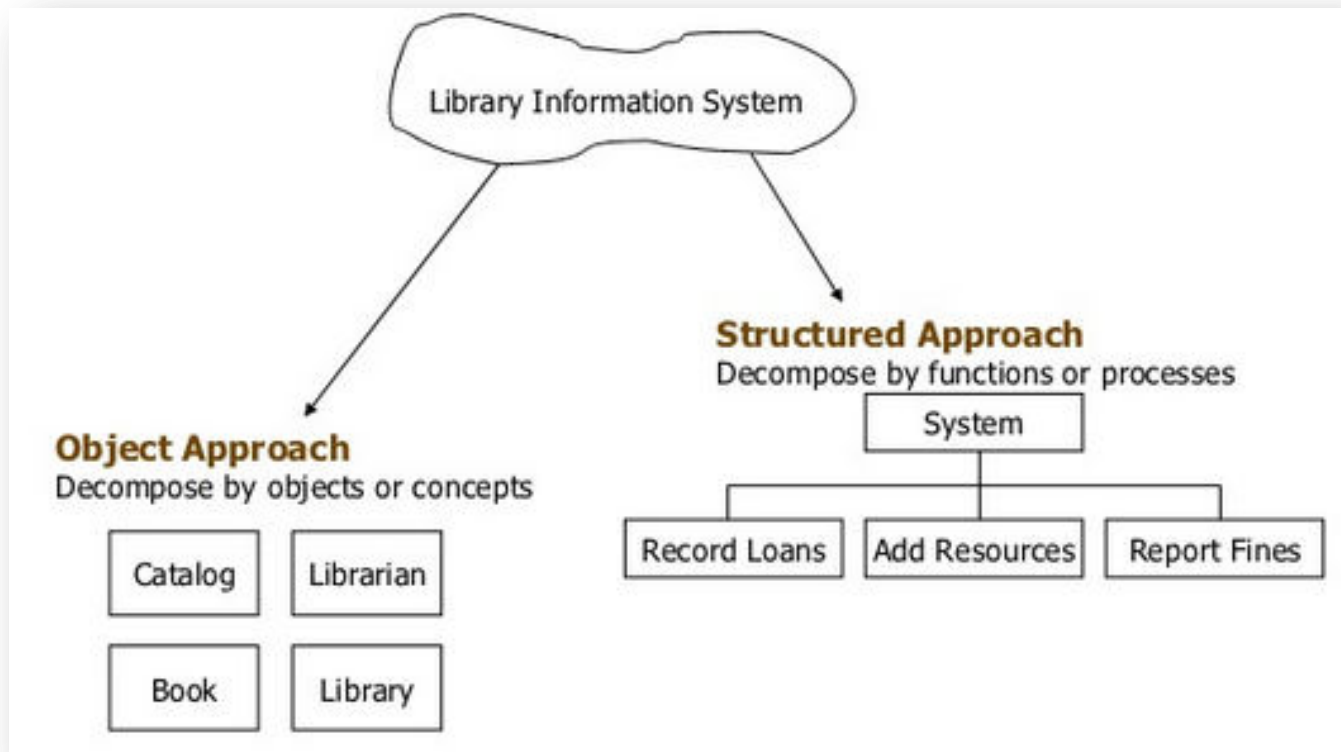
Object Oriented Paradigm

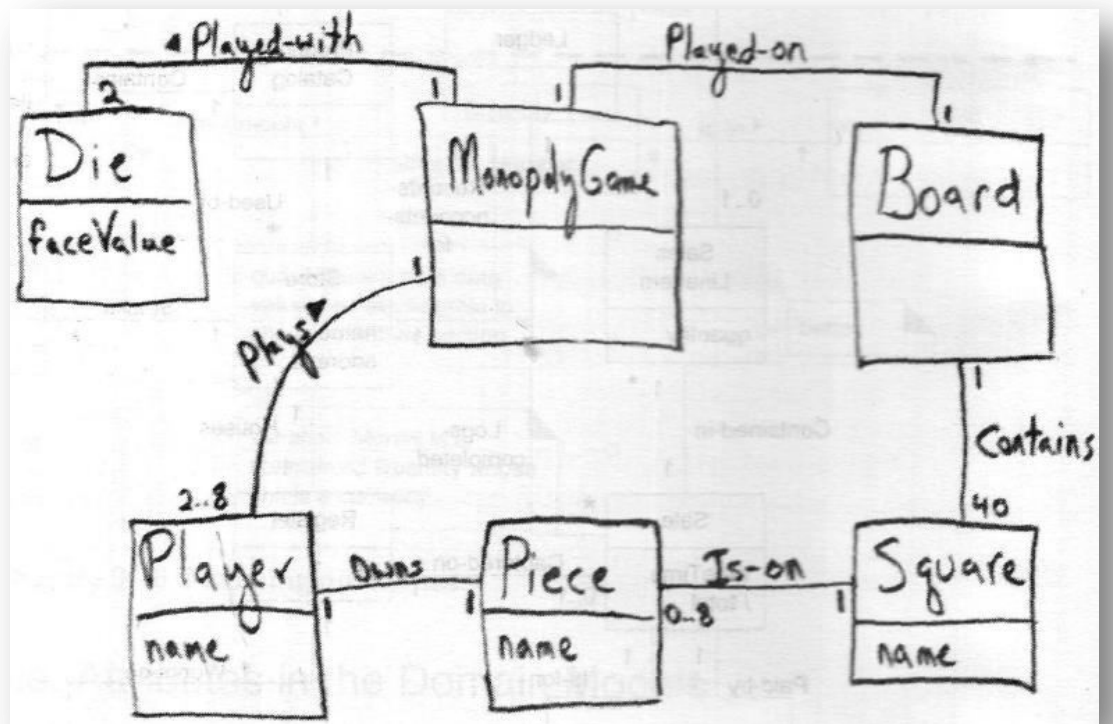
- Object Oriented is both programming and analysis/design paradigm
 - “*A paradigm is a set of theories, standards, and methods that together represent a way of organizing knowledge; that is, a way of viewing the world [Kuhn 70]*”
- Do you know any other programming paradigms?
 - Procedural [Pascal, C]
 - Logic [Prolog]
 - Functional [Lisp]
 - Object- Oriented (C++, Java, C#, VB.NET)
- Do you know any analysis/design paradigm?
 - Structural (process modeling, data flow diagrams, logic modeling)

Object Oriented vs. Functional Paradigm

- ✓ **Object Oriented Approach** – decompose* by objects or concepts
- ✓ **Structured Approach** – decompose* by functions or processes

**Decomposition is the primary strategy to deal with the complexity of S/W project.*





Advantages of Object Oriented Approach

- **Closer to how humans think naturally**

- Objects as opposed to functions e.g., a Bank

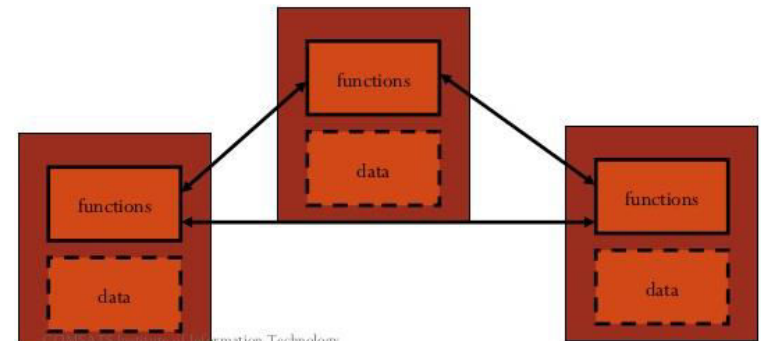
- **Complex systems become easier to understand due to better representation**

- Closer representation of reality

- **Easy to maintain due to object oriented principles such as encapsulation, inheritance, etc.**

Separation: objects interact with each other only via their membership functions which helps to maintain the integrity of the entire program.

Loose Coupling - ?



© 2005, Pearson Education, Inc. Information Technology
Object Oriented Programming

What is Object-Orientation

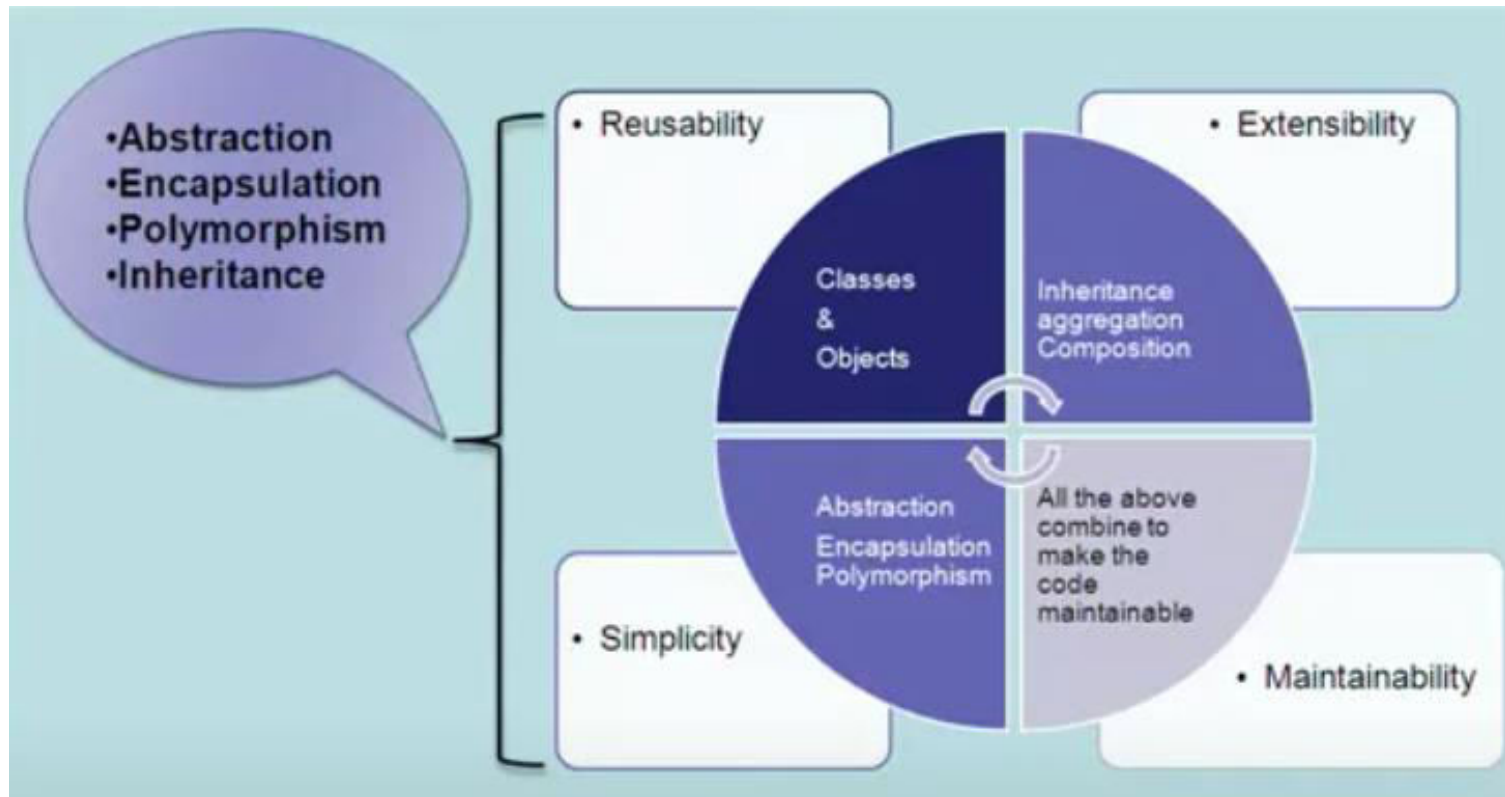
- Software is organized as a collection of discrete objects that incorporate both data structure and behavior.
- In general it includes-
 - Abstraction
 - Encapsulation
 - Inheritance
 - Polymorphism

Object-Oriented Programming

Object-oriented programming is a programming methodology characterized by the following concepts:

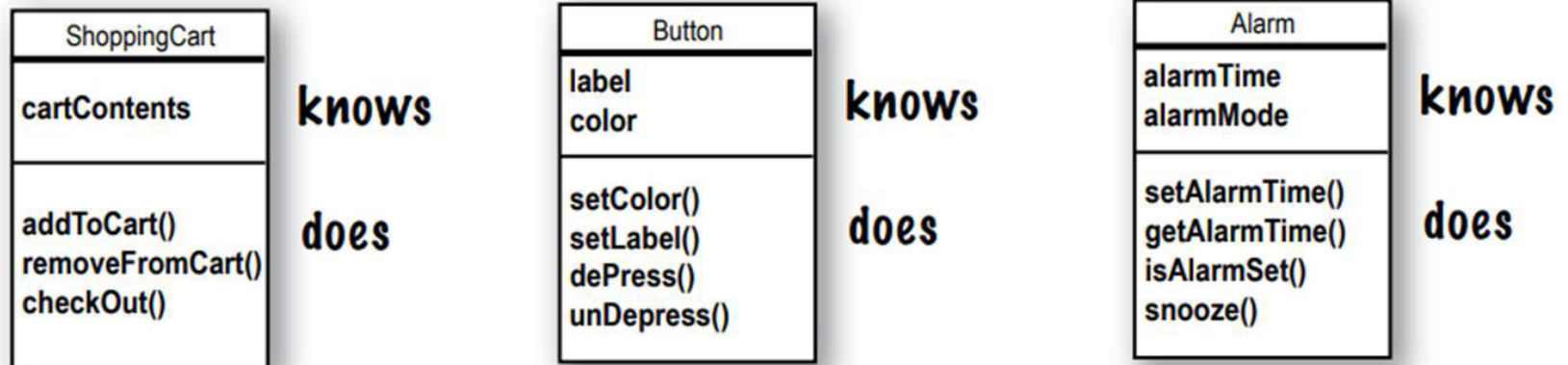
- 1. Data Abstraction:** Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.
- 2. Encapsulation:** the wrapping up of data and functions into a single unit.
- 3. Polymorphism:** Generally, the ability to appear in many forms.
 - More specifically, in OOP, it is the ability to redefine methods for derived classes
 - Ability to process objects differently depending on their data type or class
 - Giving different meanings to the same thing
- 4. Inheritance:** The process of creating new classes, called derived classes, from existing classes or base classes creating a hierarchy of parent classes and child classes. The child classes inherit the attributes and behavior of the parent classes.

Advantages of OOP Principles



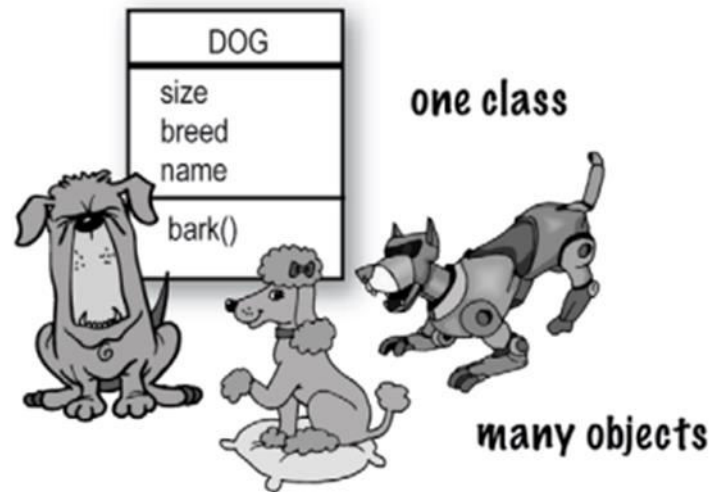
Classes and Objects - Revisit

- A class is simply a representation of a type of object.
- A blueprint that describes the details of an Object
- A class contains Member Variables (state), and Methods (Behavior)



Classes and Objects - Revisit

- An instance of a class is known as an object.
- Using that object we can access the properties and methods of a class.
- The set of activities that the object performs defines the object's behavior.
- For example a Student (object) can provide the name or address.
- In pure OOP terms an object is an **instance** of a class.

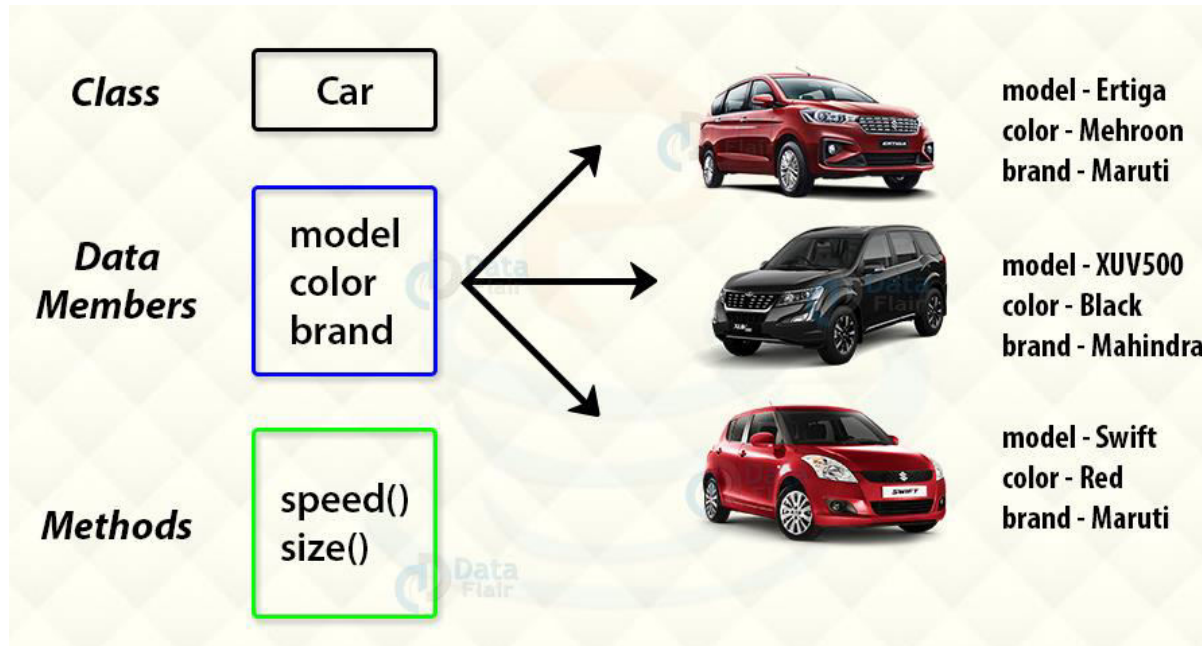


Classes and Instances

- In general, something should be a **class** if it could have instances.
- In general, something should be an **instance** if it is clearly a single member of the set defined by a class.



Another Example



O-O Principles and C++ Constructs

O-O Concept

Abstraction

Encapsulation

Information Hiding

Polymorphism

Inheritance

C++ Construct(s)

Classes

Classes

Public and Private Members

Operator overloading, Virtual functions

Derived Classes

Create a class

[visibility] [keyword class] [class name]

```
class Employee {  
    String name;  
    String ssn;  
    String emailAddress;  
    int yearOfBirth;  
  
    void print() {  
        System.out.println("Name: " + name);  
        System.out.println("SSN: " + ssn);  
        System.out.println("Email Address: " + emailAddress);  
        System.out.println("Year Of Birth: " + yearOfBirth);  
    }  
}
```

Data Values/ variables/ Attributes

Methods/ functions/ procedure

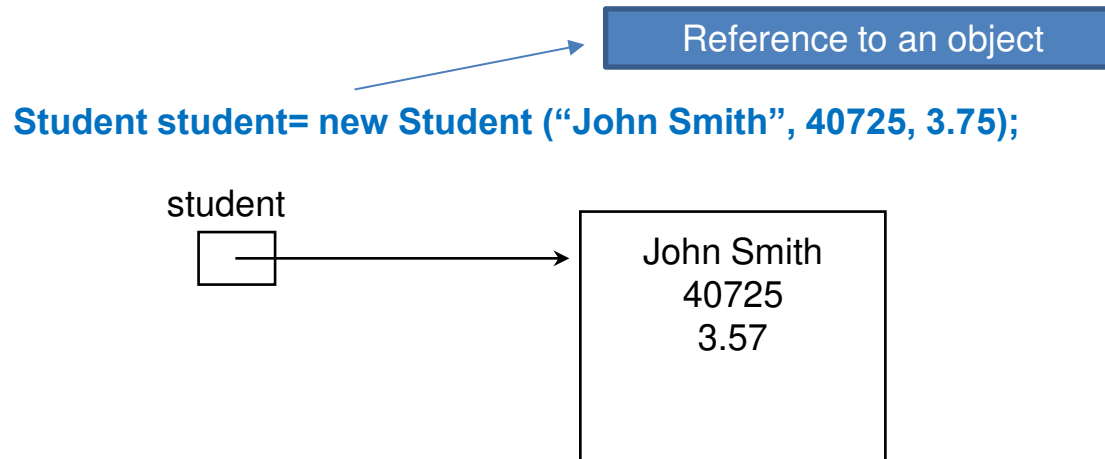
Create an Object

```
public class EmployeeDemo {  
    public static void main(String[] args) {  
        Employee e1 = new Employee();  
        e1.name = "John";  
        e1.ssn = "555-12-345";  
        e1.emailAddress = "john@company.com";  
  
        Employee e2 = new Employee();  
        e2.name = "Tom";  
        e2.ssn = "456-78-901";  
        e2.yearOfBirth = 1974;  
  
        e1.print();  
        e2.print();  
    }  
}
```

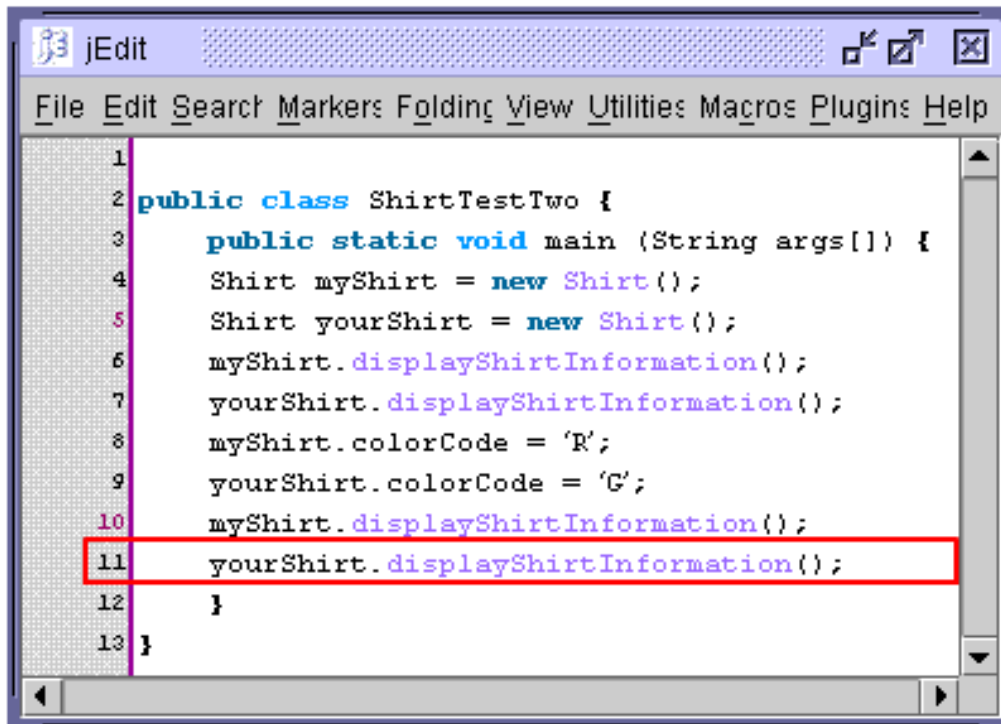
The Java **new** keyword is used to create an instance of the class.

Object References

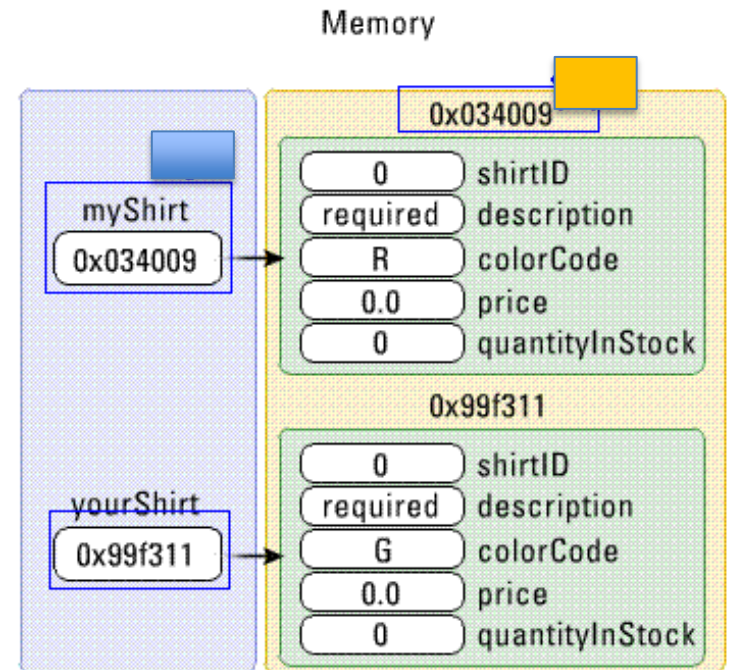
- Recall that an ***object reference*** is a variable that stores the address of an object
- A reference can also be called a *pointer*
- They are often depicted graphically:



Example



```
1
2 public class ShirtTestTwo {
3     public static void main (String args[]) {
4         Shirt myShirt = new Shirt();
5         Shirt yourShirt = new Shirt();
6         myShirt.displayShirtInformation();
7         yourShirt.displayShirtInformation();
8         myShirt.colorCode = 'R';
9         yourShirt.colorCode = 'G';
10        myShirt.displayShirtInformation();
11        yourShirt.displayShirtInformation();
12    }
13 }
```



No argument Constructors

- As the name specifies the no argument constructors of Java does not accept any parameters instead, using these constructors the instance variables of a method will be initialized with fixed values for all objects.

```
Public class MyClass {  
    Int num;  
    MyClass() {  
        num = 100;  
    }  
}
```

```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass();  
        MyClass t2 = new MyClass();  
        System.out.println(t1.num + " " + t2.num);  
    }  
}
```

Output:

```
100 100
```

Parameterized Constructors

- Most often, you will need a constructor that accepts one or more parameters. Parameters are added to a constructor in the same way that they are added to a method, just declare them inside the parentheses after the constructor's name.

```
// A simple constructor.  
class MyClass {  
    int x;  
  
    // Following is the constructor  
    MyClass(int i ) {  
        x = i;  
    }  
}
```

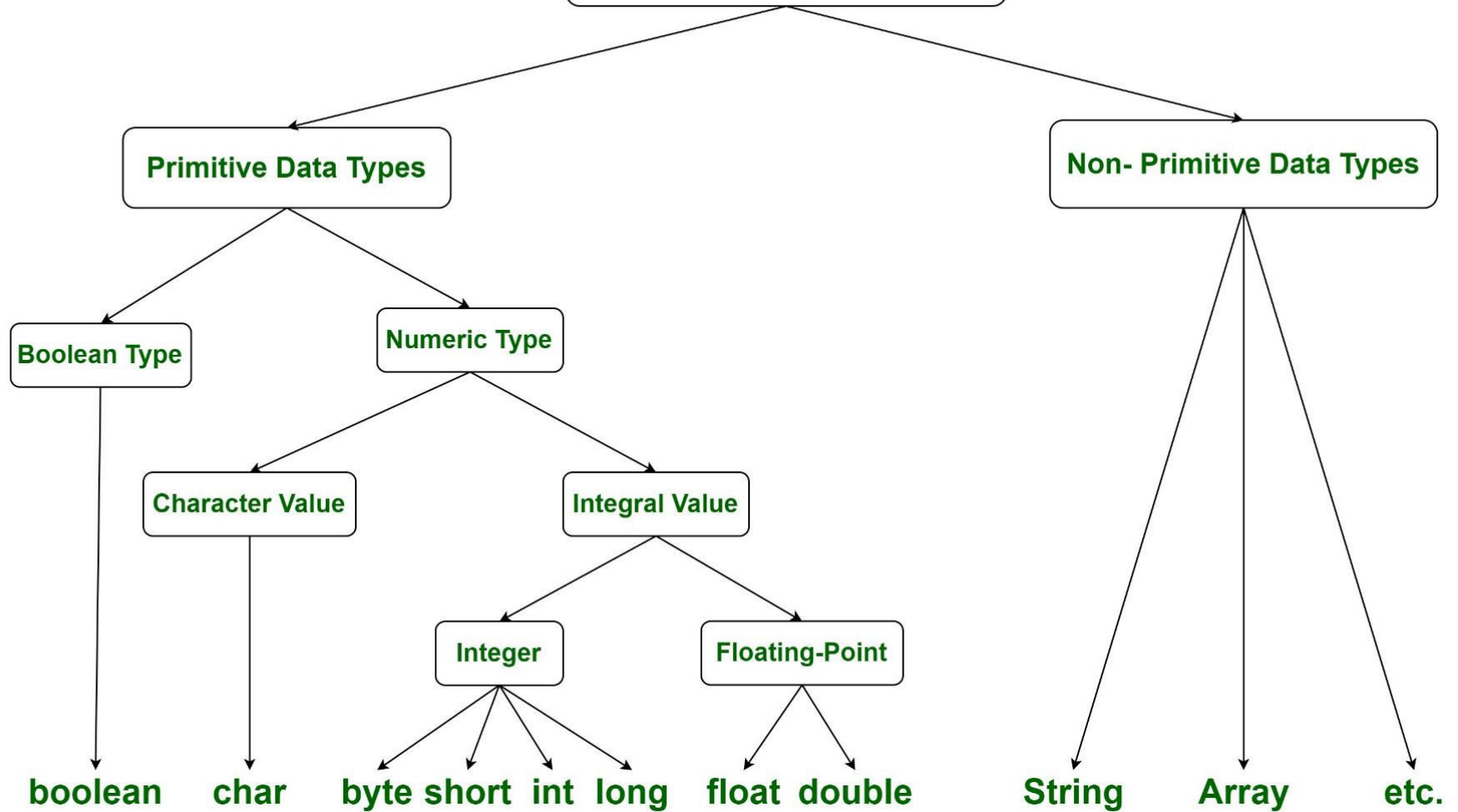
```
public class ConsDemo {  
    public static void main(String args[]) {  
        MyClass t1 = new MyClass( 10 );  
        MyClass t2 = new MyClass( 20 );  
        System.out.println(t1.x + " " + t2.x);  
    }  
}
```

Output: 10 20

Data types in Java

- Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:
 1. **Primitive Data Types:** A primitive data type is **pre-defined by the programming language**. The size and type of variable values are specified, and it has no additional methods. The primitive data types include boolean, char, byte, short, int, long, float and double.
 2. **Non-Primitive Data Types:** These data types are not actually defined by the programming language but are **created by the programmer**. They are also called “reference variables” or “object references” since they reference a memory location which stores the data. The non-primitive data types include Classes, Interfaces, and Arrays.

Data Types in Java



Visibility

1. **Private:** The access level of a private modifier is **only within the class**. It cannot be accessed from outside the class.
2. **Default:** The access level of a default modifier is **only within the package**. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected:** The access level of a protected modifier is **within the package and outside the package through child class**. If you do not make the child class, it cannot be accessed from outside the package.
4. **Public:** The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Types of Variables

- There are three kinds of variables in Java:
 - Local variables
 - Instance variables
 - Class/Static variables

Example

```
public class Test {  
    public void pupAge() {  
        int age = 0;  
        age = age + 7;  
        System.out.println("Puppy age is : " + age);  
    }  
  
    public static void main(String args[]) {  
        Test test = new Test();  
        test.pupAge();  
    }  
}
```

Scope of local variable

Important to initialize

Cannot use 'age' here

Output:

Puppy age is: 7

Here, **age** is a local variable. This is defined inside *pupAge()* method and its scope is limited to only this method.

Instance Variable Example

Declared outside the method

```
public class Employee {  
  
    // this instance variable is visible for any child class.  
    public String name;  
  
    // salary variable is visible in Employee class only.  
    private double salary;  
  
    // The name variable is assigned in the constructor.  
    public Employee (String empName) {  
        name = empName;  
    }  
  
    // The salary variable is assigned a value.  
    public void setSalary(double empSal) {  
        salary = empSal;  
    }  
  
    // This method prints the employee details.  
    public void printEmp() {  
        System.out.println("name : " + name );  
        System.out.println("salary : " + salary);  
    }  
  
    public static void main(String args[]) {  
        Employee empOne = new Employee("Ransika");  
        empOne.setSalary(1000);  
        empOne.printEmp();  
    }  
}
```

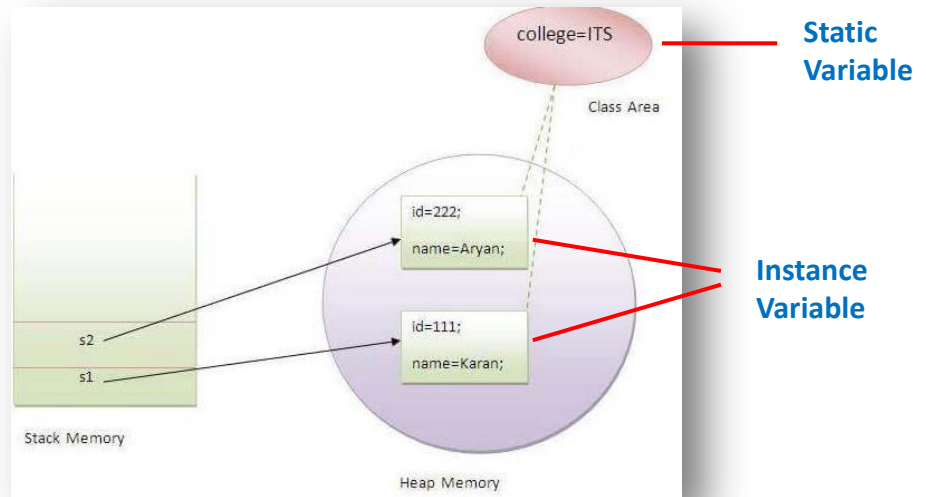
Output:

```
name : Ransika  
salary :1000.0
```

Example Static Variable

```
class Student{
    int rollno;//instance variable
    String name;
    static String college ="ITS";//static variable
    //constructor
    Student(int r, String n){
        rollno = r;
        name = n;
    }
    //method to display the values
    void display (){System.out.println(rollno+" "+name+" "+college);}
}

//Test class to show the values of objects
public class TestStaticVariable1{
    public static void main(String args[]){
        Student s1 = new Student(111,"Karan");
        Student s2 = new Student(222,"Aryan");
        //we can change the college of all objects by the single line of code
        //Student.college="BBDIT";
        s1.display();
        s2.display();
    }
}
```

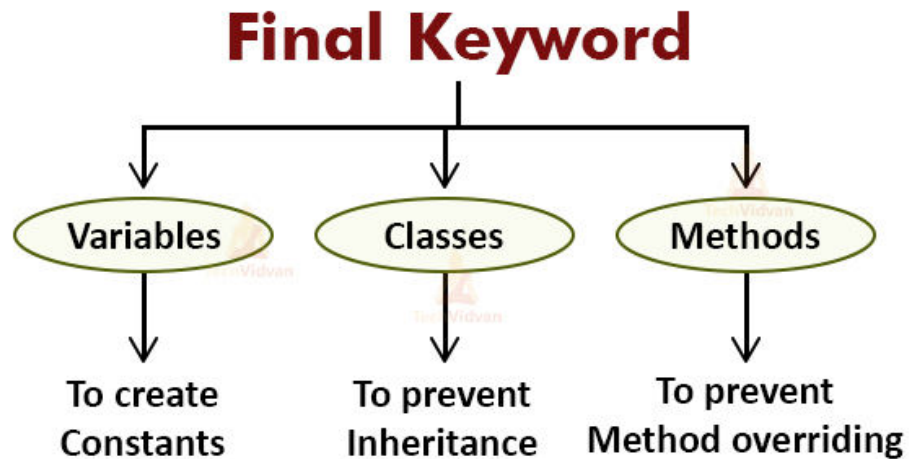


Output:

```
111 Karan ITS
222 Aryan ITS
```

Final Keyword

The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context.



Example

```
class A {  
    final int a;  
    void f(final int b) {  
        a=2; b=5;  
    }  
}
```

final Variable
Can't be Modified

```
final class A {  
class B extends A {
```

final class
Can't be Extended

```
class A {  
    final void f() {  
    }  
class B extends A {  
    void f() {  
    }  
}
```

final method
Can't be Overridden

Final method is inherited but you cannot override it

```
class Bike{
    final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
    public static void main(String args[]){
        new Honda2().run();
    }
}
```

Blank final variable (AN CARD number of an employee)

```
class Student{
    int id;
    String name;
    final String PAN_CARD_NUMBER;
    ...
}
```

Initialize blank final variable

```
class Bike10{
    final int speedlimit;//blank final variable

    Bike10(){
        speedlimit=70;
        System.out.println(speedlimit);
    }

    public static void main(String args[]){
        new Bike10();
    }
}
```

static blank final variable

```
class A{
    static final int data;//static blank final variable
    static{ data=50;}
    public static void main(String args[]){
        System.out.println(A.data);
    }
}
```

Instance
Method

Vs.

Static
Method

```
class Employee
{
    int empId;
    String empName;
    static String companyName = "TCS";
    //static method to valueChange the value of static variable
    static void valueChange()
    {
        companyName = "DataFlair";
    }
    //constructor to initialize the variable
    Employee(int id, String name){
        empId = id;
        empName = name;
    }
    //method to display values
    void display()
    {
        System.out.println(empId+" "+empName+" "+companyName);
    }
}
//class to create and display the values of object
public class Demo
{
    public static void main(String args[])
    {
        Employee.valueChange();//calling valueChange method
        //creating objects
        Employee EmployeeObj = new Employee(218,"Kushal");
        Employee EmployeeObj1 = new Employee(635,"Bhumika");
        Employee EmployeeObj2 = new Employee(147,"Renuka");
        //calling display method
        EmployeeObj.display();
        EmployeeObj1.display();
        EmployeeObj2.display();
    }
}
```

Instance Variable

Static Variable

Static Method

Parameterized constructor

Instance method

Static method Calling with the
reference of class

Instance method Calling with the
reference of class object

Class Access Level

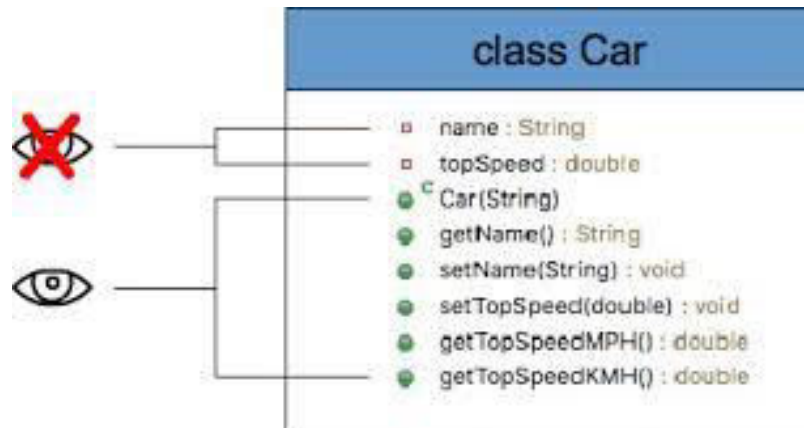
- A top-level class as private would be completely useless because nothing would have access to it.
- Java doesn't allow a top level class to be private. Only 'public' or 'package'.
- **Private classes are allowed**, but only as inner or nested classes. If you have a private inner or nested class, then access is restricted to the scope of that outer class.

Cont..

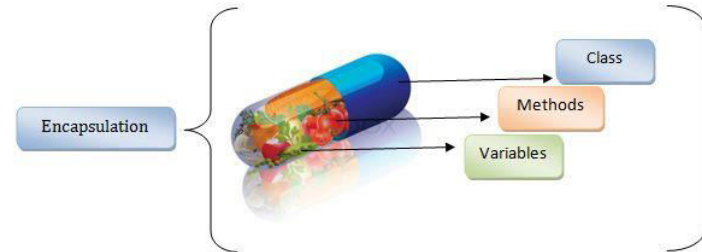
- We can not declare an outer class as private. More precisely, we can not use the private access specifier with an outer class.
- As soon as you try to use the private access specifier with a class you will get a message in Eclipse as the error that only public, final, and abstract can be used as an access modifier with a class.
- You have only two options, **public** or no access modifier at all. By omitting public you, implicitly, limit class access to within the package (aka: package-private).

Encapsulation

- a.k.a. information hiding
- Objects encapsulate:
 - **property**
 - **behavior** as a collection of methods invoked by messages
 - **state** as a collection of instance variables



Encapsulation - Continued



- A concept of ‘**Self-containing**’
- Information hiding
 - ‘internal’ structure is hidden from their surroundings
- Behavior and information is represented or **implemented internally**
- Functionality and behavior characterized by **operations**
- "Encapsulation is a process of binding data members (variables and properties) and member functions (methods) into a single unit".
- And a class is the best example of encapsulation.

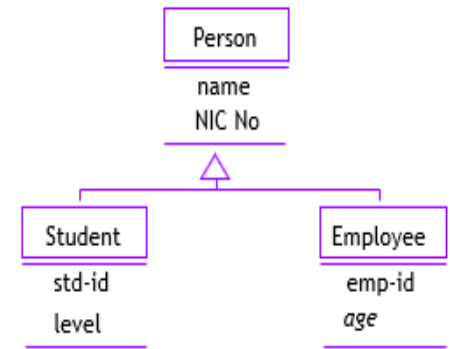
How Encapsulation is achieved in a class

- To do this:
 1. Make all the data members private.
 2. Create public setter and getter functions for each data member in such a way that the set function set the value of data member and get function get the value of data member.

Advantages of Encapsulation

- **Data Hiding:** The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables.
- **Increased Flexibility:** We can make the variables of the class as read-only or write-know that only depending on our requirement.
- **Reusability:** Encapsulation also improves the re-usability and easy to change with new requirements.
- **Testing code is easy:** Encapsulated code is easy to test for unit testing.

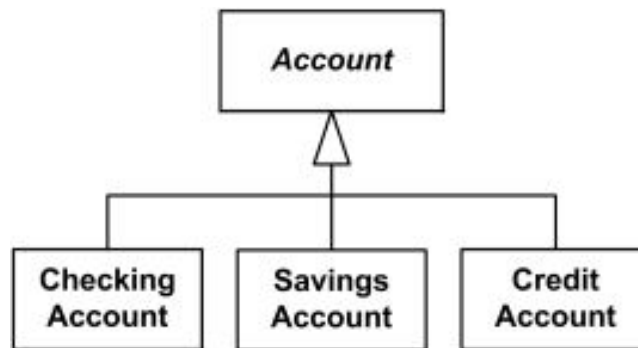
Inheritance



- Inheritance is a mechanism in which one class acquires the property of another class.
- It is a mechanism where you can derive a class from another class for a hierarchy of classes that share a set of attributes and methods.
- With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability.
 - **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
 - **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.

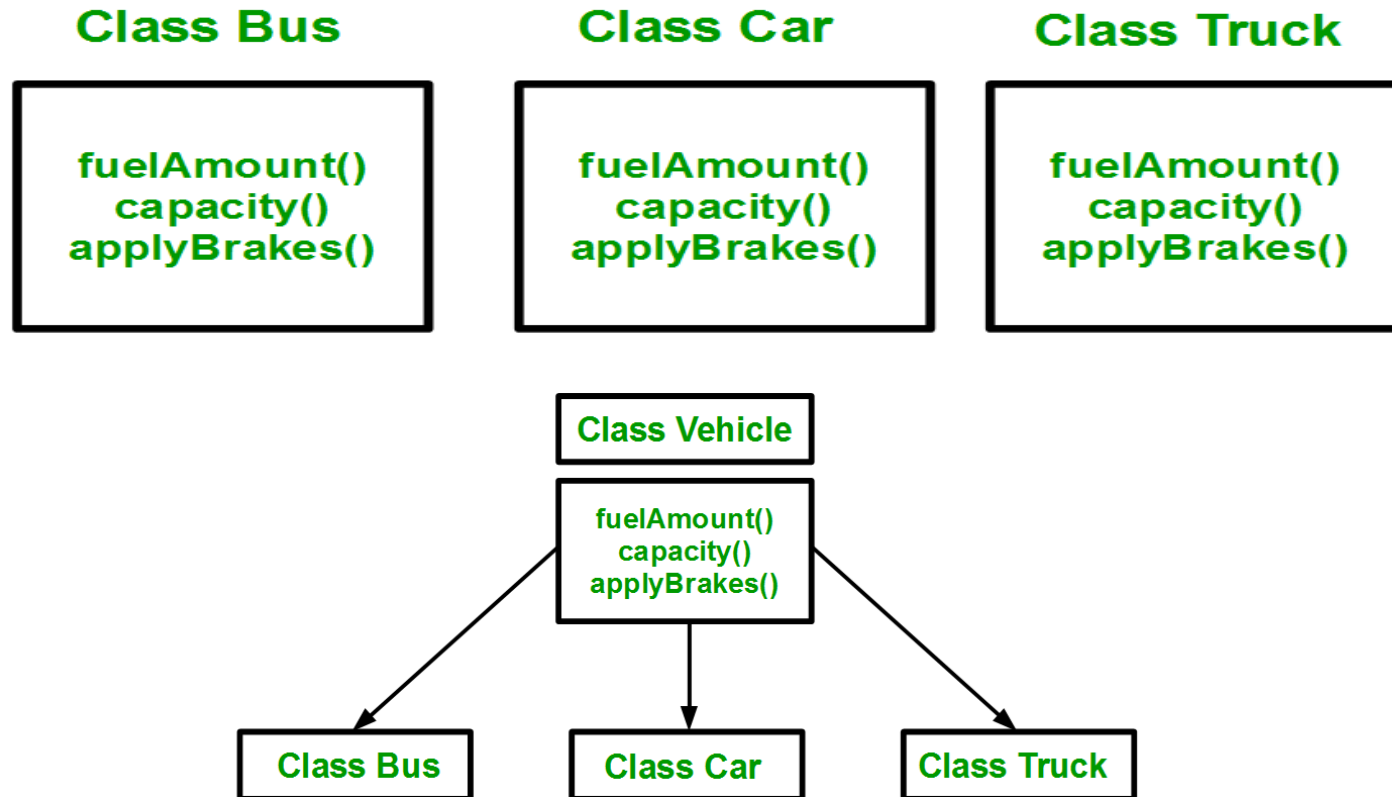
Inheritance

- Inheritance enables you to create a new class that inherits the properties from ***another class*** or ***base class*** and the class that inherits those members is called the ***derived class***.
- So the derived class has the properties of the base class and its own class properties as well.



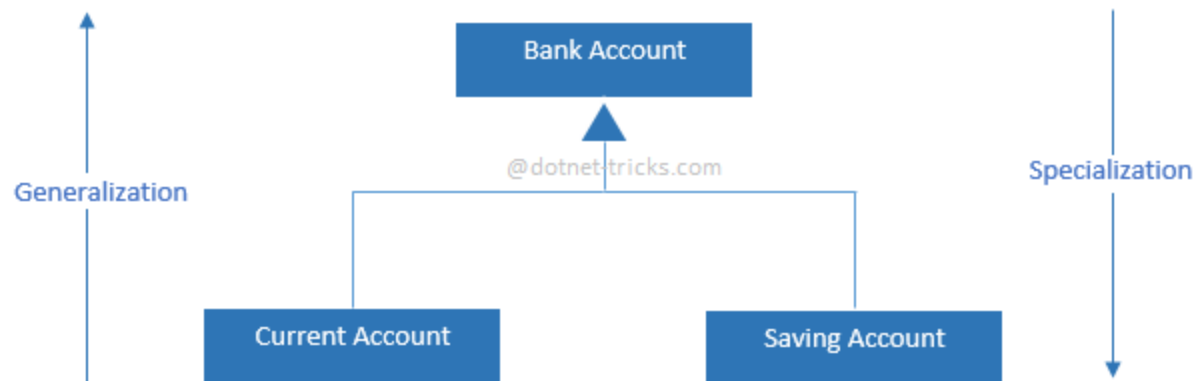
Base Class: Account
Derived Class(es): Checking, Saving, Credit

When and Why to use Inheritance?



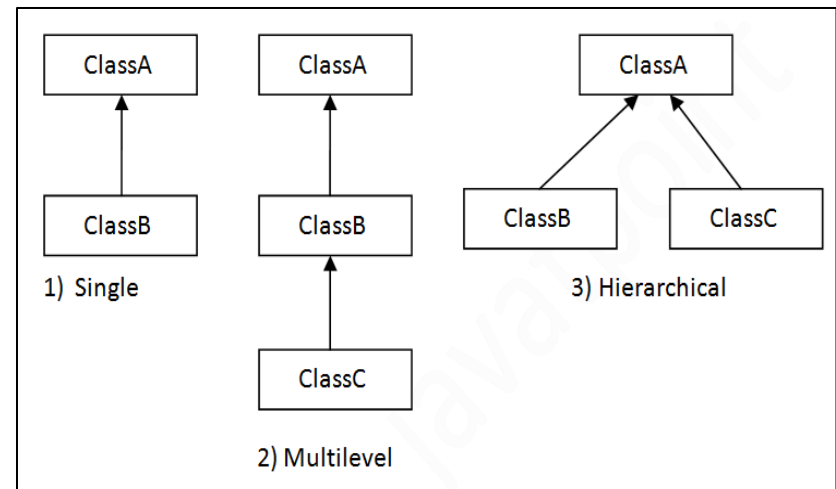
Generalization and Specialization

- In general terms, **generalization** and **specialization** both refers to **inheritance** but the approach in which they are implemented are different.
- **Generalization:**
 - Generalization is the process of extracting common characteristic from two or more classes and combining them into a generalized super class. Common characteristic can be attributes or behavior
- **Specialization:**
 - Specialization is the reverse process of Generalization means creating new sub classes from an existing class.



Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance



Single Inheritance

When a class inherits another class, it is known as a single inheritance.

```
class Animal{  
    void eat(){System.out.println("eating...");}  
}  
class Dog extends Animal{  
    void bark(){System.out.println("barking...");}  
}  
class TestInheritance{  
    public static void main(String args[]){  
        Dog d=new Dog();  
        d.bark();  
        d.eat();  
    }}  

```

Multilevel Inheritance

When there is a chain of inheritance, it is known as multilevel inheritance.

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class BabyDog extends Dog{
void weep(){System.out.println("weeping...");}
}
class TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```


Hierarchical Inheritance

When two or more classes inherits a single class, it is known as *hierarchical inheritance*.

```
class Animal{
    void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
    void bark(){System.out.println("barking...");}
}
class Cat extends Animal{
    void meow(){System.out.println("meowing...");}
}
class TestInheritance3{
    public static void main(String args[]){
        Cat c=new Cat();
        c.meow();
        c.eat();
        //c.bark();//C.T.Error
    }}
}
```

Constructors and Initialization

- Classes use constructors to initialize instance variables
 - When a subclass object is created, its constructor is called.
 - It is the responsibility of the subclass constructor to invoke the appropriate superclass constructors so that the instance variables defined in the superclass are properly initialized
- **Superclass constructors can be called using the "super" keyword** in a manner similar to "this"
 - It must be the first line of code in the constructor
- If a call to super is not made, the system will **automatically** attempt to invoke the no-argument constructor of the superclass.

Use of Super Keyword

- super keyword is similar to this keyword in Java.
- It is used to refer to the immediate parent class of the object.
- **super ()** calls the parent class constructor with no argument.
- **super.methodname** calls method from parents class.
- It is used to call the parents class variable.

Constructors - Example

```
public class BankAccount
{
```

```
    private String ownersName;
    private int accountNumber;
    private float balance;
```

BASE Class
Constructor

```
    public BankAccount(int anAccountNumber, String aName)
```

```
    {
        accountNumber = anAccountNumber;
        ownersName = aName;
    }
    [...]
}
```

```
public class OverdraftAccount extends BankAccount
{
```

```
    private float overdraftLimit;
```

```
    public OverdraftAccount(int anAccountNumber, String aName, float
aLimit)
```

```
    {
        super(anAccountNumber, aName);
        overdraftLimit = aLimit;
    }
}
```

DERIVED Class
Constructor

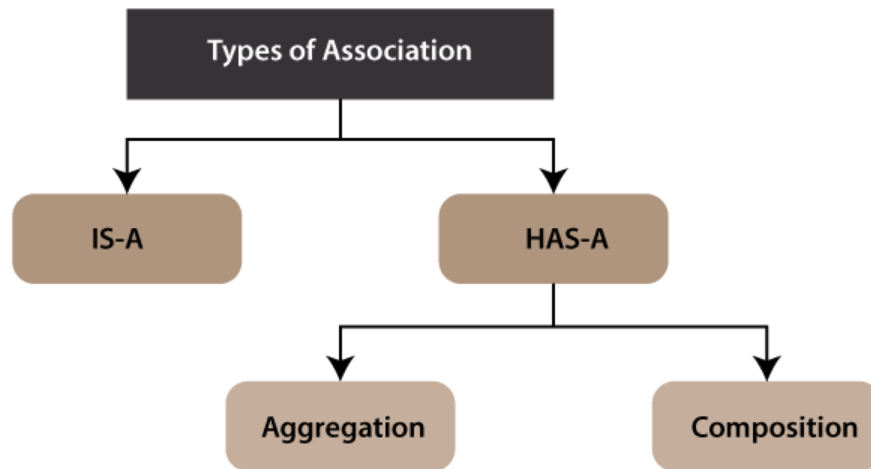
The protected Modifier

- Visibility modifiers determine which class members are inherited and which are not
- Variables and methods declared with `public` visibility are inherited; those with `private` visibility are not
- But `public` variables violate the principle of encapsulation
- There is a third visibility modifier that helps in inheritance situations: `protected`

The protected Modifier

- The `protected` modifier allows a member of a base class to be inherited into a child
- Protected visibility provides more encapsulation than public visibility does
- However, protected visibility is not as tightly encapsulated as private visibility

Relationships among Classes



Advantages:

- Code Reusability
- Cost Effective
- Reduce Redundancy

- Defines a connection or relation between two separate classes that are set up through their objects.
- Association relationship indicates how objects know each other and how they are using each other's functionality.
- It can be one-to-one, one-to-many, many-to-one and many-to-many.

Types of Association	Example
One-to-One	A person can have only one passport
One-to-Many	A bank can have many employees
Many-to-One	Multiple cities belong to one state
Many-to-Many	Multiple students can be associated with a single teacher and a single student can also be associated with multiple teachers

Aggregation

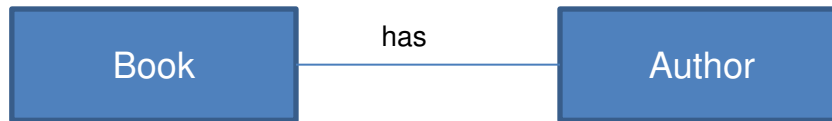
- Aggregation in Java is a special type of association. It has the following characteristics:
- It represents the Has-A relationship.
- *Aggregation in Java follows a one-way or one-to-one relationship.*
- Ending one entity won't affect another, both can be present independently.

Composition

- This is a restricted form of Java aggregation that is the quantities are highly dependent on each other
- It represents a part-of relationship. One entity cannot exist without the other.
- *Composition in Java represents a one-to-many relationship.*

One-to-One Association

- A book has author.



Implementation

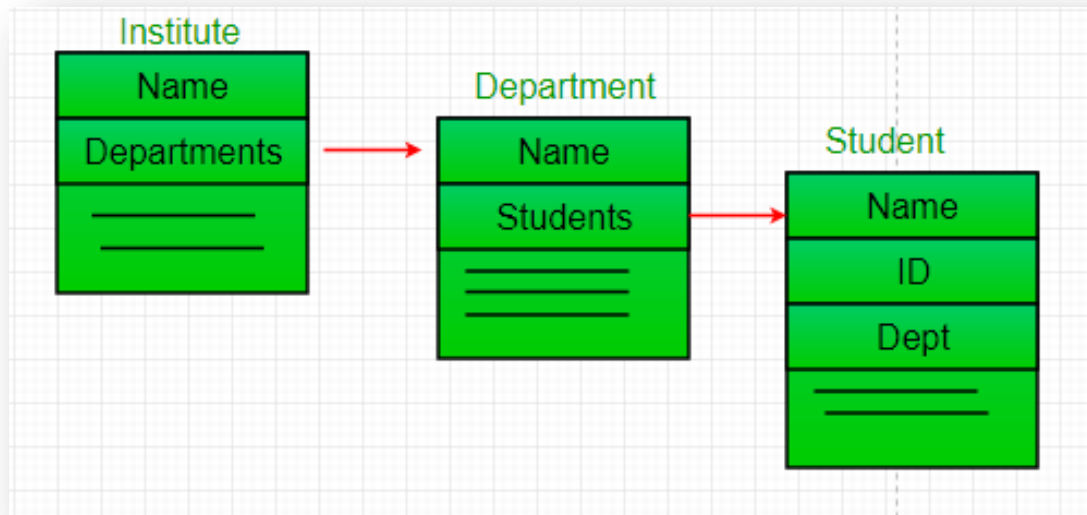
```
class Author
{
    String authorName;
    int age;
    String place;

    // Author class constructor
    Author(String name, int age, String place)
    {
        this.authorName = name;
        this.age = age;
        this.place = place;
    }
}
```

```
class Book
{
    String name;
    int price;
    // author details
    Author author;
    Book(String n, int p, Author author)
    {
        this.name = n;
        this.price = p;
        this.author = author;
    }

    public static void main(String[] args) {
        Author author = new Author("John", 42, "USA");
        Book b = new Book("Java for Begginer", 800, author);
        System.out.println("Book Name: "+b.name);
        System.out.println("Book Price: "+b.price);
        System.out.println("-----Auther Details-----");
        System.out.println("Auther Name: "+b.author.authorName);
        System.out.println("Auther Age: "+b.author.age);
        System.out.println("Auther place: "+b.author.place);
    }
}
```

In this example, there is an Institute which has no. of departments like CSE, EE. Every department has no. of students.



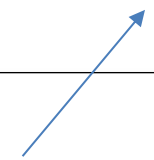
Student Class

```
public class Student {  
    String name;  
    int id ;  
    String dept;  
  
    Student(String name, int id, String dept)  
    {  
  
        this.name = name;  
        this.id = id;  
        this.dept = dept;  
    }  
}
```

Department class contains a list of Students

Department Class

```
public class Department {  
    String name;  
    private List<Student> students;  
    Department(String name, List<Student> students)  
    {  
        this.name = name;  
        this.students = students;  
    }  
  
    public List<Student> getStudents()  
    {  
        return students;  
    }  
}
```



Institute class contains a list of departments

Institute Class

```
public class Institute {
    String instituteName;
    private List<Department> departments;

    Institute(String instituteName, List<Department> departments)
    {
        this.instituteName = instituteName;
        this.departments = departments;
    }

    // count total students of all departments in a given institute
    public int getTotalStudentsInInstitute()
    {
        int noOfStudents = 0;
        List<Student> students;
        for(Department dept : departments)
        {
            students = dept.getStudents();
            for(Student s : students)
            {
                noOfStudents++;
            }
        }
        return noOfStudents;
    }
}
```

Main Class

```
public class AssociationDemo {

    public static void main (String[] args)
    {
        Student s1 = new Student("Ahmed", 1, "SE");
        Student s2 = new Student("Ali", 2, "SE");
        Student s3 = new Student("Rehmat", 1, "EE");
        Student s4 = new Student("Moosa", 2, "EE");

        List <Student> se_students = new ArrayList<Student>();
        se_students.add(s1);
        se_students.add(s2);

        List <Student> ee_students = new ArrayList<Student>();
        ee_students.add(s3);
        ee_students.add(s4);

        Department SE = new Department("SE", se_students);
        Department EE = new Department("EE", ee_students);

        List <Department> departments = new ArrayList<Department>();
        departments.add(SE);
        departments.add(EE);

        Institute institute = new Institute("FAST", departments);

        System.out.print("Total students in institute: ");
        System.out.print(institute.getTotalStudentsInInstitute());
    }
}
```

Why Inheritance

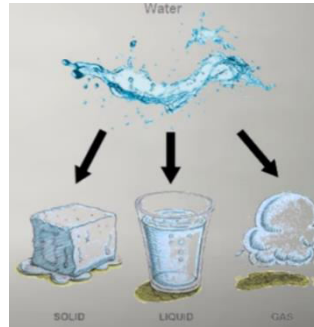
- Show **similarities**
- **Extensibility**
- ‘**Software Reuse**’
- Easy **modification** of model by performing modification in one place
- Avoid **redundancy**, leading to smaller and more **efficient** model, easier to understand

Polymorphism

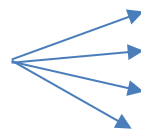
Combination of two Greek words

- Poly (**many**) morphism (**form**)

1. Water -> Solid, Liquid, Gas



2. For example, A Person



In shopping mall, behaves like a customer

In metro bus, behaves like a passenger

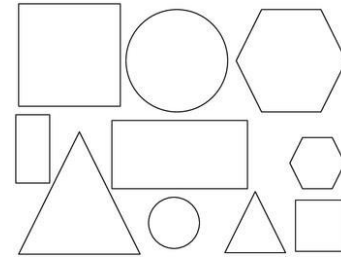
In university, behaves like a student

In home, behaves like a daughter/son

Same person has different behavior in different situation.

Cont..

3. Shapes-> Circle ,Square, Triangle, Rectangle



- More than one function with same name but with different working.

In Java, we can achieve polymorphism through function/methods.

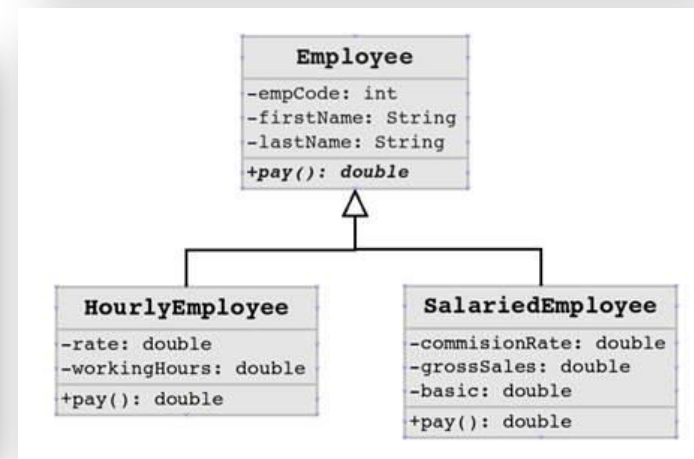
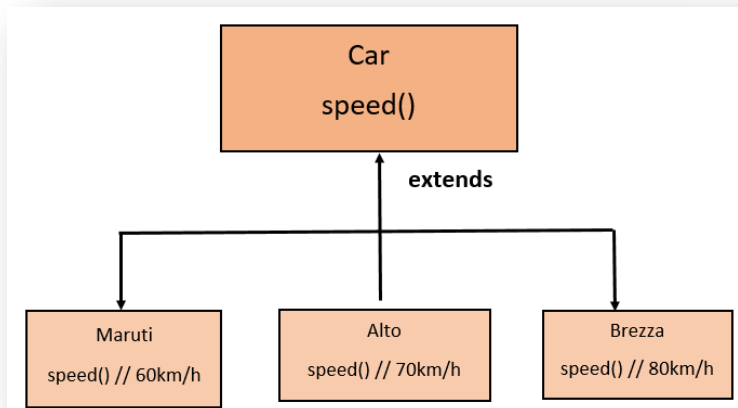
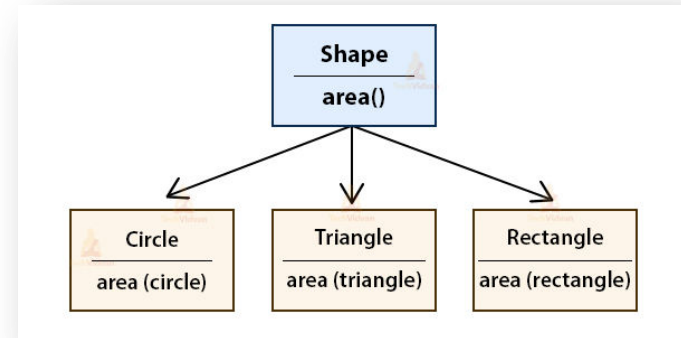
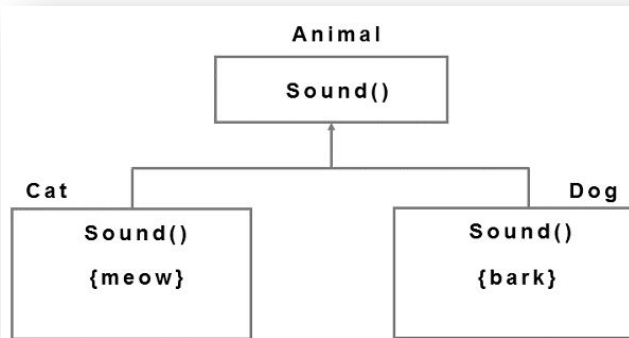
Cont..

- The ability to hide many different implementations behind a single interface.
- One type of operation can be implemented in different ways by different classes.
- In java, the method "***println()***", single name many forms. It is an example of polymorphism.

```
println (10); // int Literal  
println ("Hello Java"); // String  
println (23.4); // Double
```

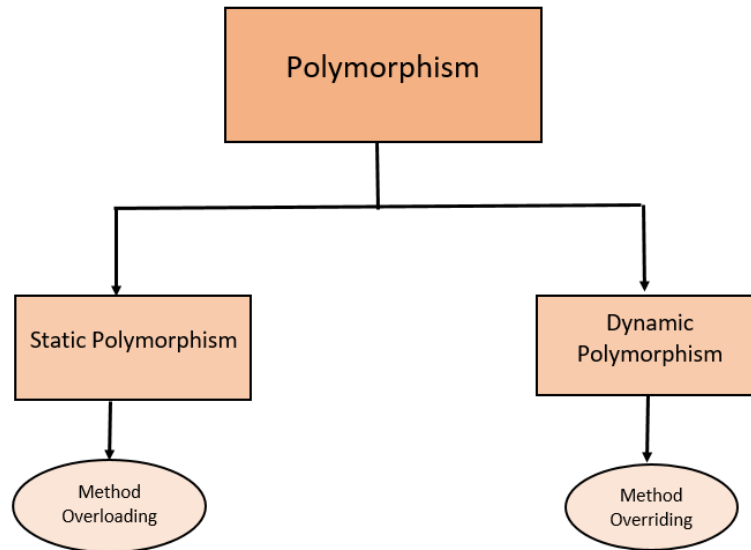
Polymorphism reduces the complexity

Examples



Types of Polymorphism

- In Java Polymorphism is mainly divided into two types:-
 - Compile time Polymorphism (early binding/ Static Polymorphism)
 - Runtime Polymorphism (late binding/ Dynamic Polymorphism)



Comparison of Method Overloading and Method Overriding

Method Overloading	Method Overriding
1. Same Name of Methods	1. Same Name of Methods
2. In Same Class	2. Must be in different classes
3. Different arguments <ul style="list-style-type: none">• No. of arguments• Sequence of arguments• Type of argument	3. Same arguments <ul style="list-style-type: none">• No. of arguments• Sequence of arguments• Type of argument
	4. Inheritance (is-a relationship)

Compile time Polymorphism

Compile time Polymorphism: This type of Polymorphism is achieved by **method overloading**.

Method Overloading - When there are multiple methods with same name but different parameters then these methods are said to be overloaded. Methods can be overloaded by change in **number of arguments**, **sequence of arguments** or **type of arguments**.

```
int max(int a, int b) {  
    if (a >= b)  
        return a;  
    else  
        return b;  
}
```

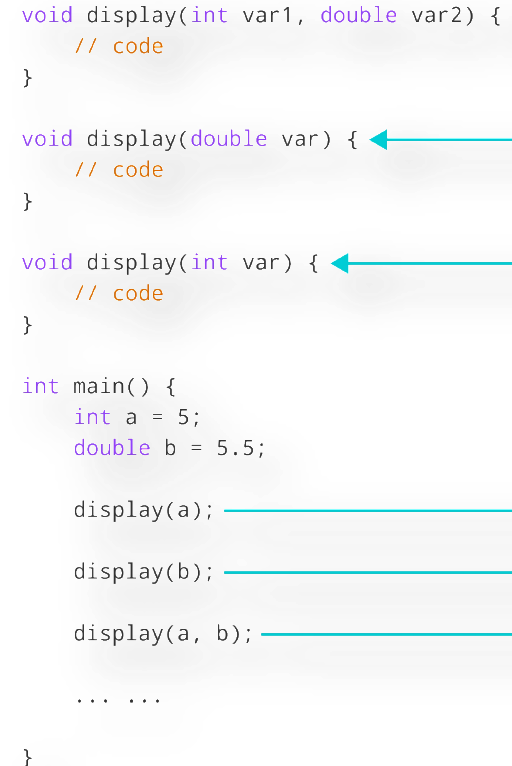
```
float max(float a, float b) {  
    if (a >= b)  
        return a;  
    else  
        return b;  
}
```

```
void myFunction()  
void myFunction(int a)  
void myFunction(float a)  
void myFunction(int a, float b)  
float myFunction (float a, int b)
```

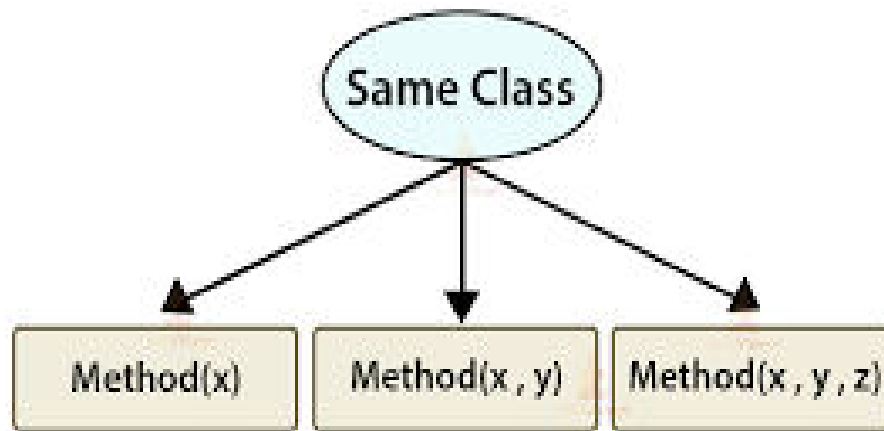
Method Overloading

- Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different.
- It is similar to [constructor overloading](#) in Java, that allows a class to have more than one constructor having different argument lists.

```
void display(int var1, double var2) {  
    // code  
}  
  
void display(double var) {  
    // code  
}  
  
void display(int var) {  
    // code  
}  
  
int main() {  
    int a = 5;  
    double b = 5.5;  
  
    display(a);  
    display(b);  
    display(a, b);  
  
    ...  
}
```



Method Overloading in Java



Three ways to overload a method

- In order to overload a method, the argument lists of the methods must differ in either of these:

1. Number of parameters.

For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.

For example:

```
add(int, int)
add(int, float)
```

Cont..

3. Sequence of Data type of parameters.

For example:

```
add(int, float)
add(float, int)
```

- **Invalid case for Method Overloading**

if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Can we overload main method in java?

- Yes, **main method** can be **overloaded**. **Overloaded main method** has to be called from inside the "public static void **main**(String args[])" as this is the entry point when the class is launched by the JVM. Also **overloaded main method** can have any qualifier as a normal **method** have.

Run time Polymorphism

- This type of polymorphism is achieved by **Function overriding**. And it is also known as **late binding** / **Dynamic binding**.
- Dynamic Polymorphism decides which method to execute in runtime.
- The compiler identifies the type of object at **run time** and then matches the function call with the correct function definition.

Method Overriding

1. Same Name of Methods
2. Must be in different classes
3. Same arguments
 - No. of arguments
 - Sequence of arguments
 - Type of argument
4. Inheritance (is-a relationship)

Method Overriding

- Method overriding allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super classes or parent classes. The implementation in subclasses overrides (replaces) the implementation in the superclass by providing a method that has the same name , same parameter or signature and same return type as the method in parent class.

Method Overriding

```
class Company {  
    public void address() {  
        System.out.println("This is Address of Crunchify Company...");  
    }  
}
```

```
class eBay extends Company {  
    public void address() {  
        System.out.println("This is eBay's Address...");  
    }  
}
```

```
public class Test {  
    public static void main(String args[])  
    {  
        Company a = new Company(); // Company reference and object  
        eBay b = new eBay(); // eBay reference and object  
        a.address(); //runs the method in Company class  
        b.address(); // runs the method in eBay class  
    }  
}
```

Method Overriding

1. Same Name of Methods
2. Must be in different classes
3. Same arguments
 - No. of arguments
 - Sequence of arguments
 - Type of argument
4. Inheritance (is-a relationship)



Output

```
This is Address of Crunchify Company...  
This is eBay's Address...
```

Cont..

```
class Company {  
    public void address() {  
        System.out.println("This is Address of Crunchify Company...");  
    }  
}
```

```
class eBay extends Company {  
    public void address() {  
        System.out.println("This is eBay's Address...");  
    }  
}
```

```
public class Test {  
    public static void main(String args[]) {  
        Company a = new Company(); // Company reference and object  
        Company b = new eBay(); // Company reference but eBay object  
  
        a.address();// runs the method in Company class  
        b.address();// Runs the method in eBay class  
    }  
}
```

Output

```
This is Address of Crunchify Company...  
This is eBay's Address...
```


Benefits of Polymorphism

- Method overloading allows methods that perform similar or closely related functions to be accessed through a common name. For example, a program performs operations on an array of numbers which can be int, float, or double type. Method overloading allows you to define three methods with the same name and different types of parameters to handle the array operations.
- Method overloading can be implemented on constructors allowing different ways to initialize objects of a class. This enables you to define multiple constructors for handling different types of initializations.
- Method overriding allows a subclass to use all the general definitions that a superclass provides and adds specialized definitions through overridden methods.
- Method overriding works together with inheritance to enable code-reuse of existing classes without the need for re-compilation.

Abstraction and Encapsulation

Abstraction	Encapsulation
Abstraction is the method of hiding the unwanted information.	Encapsulation is a method to hide the data in a single entity or unit along with a method to protect information from outside.
We can implement abstraction using abstract class and interfaces.	Encapsulation can be implemented using by access modifier i.e. private, protected and public.
Implementation complexities are hidden using abstract classes and interfaces.	the data is hidden using methods of getters and setters.

Example

How the machine is working is hidden (the details about internal data like how much cans it contains, mechanism etc.). This is called **Abstraction**.



Automatic cola vending machine

Here, Cola vending machine is a class.

It contains both

- data i.e. Cola can and
- operations i.e. service mechanism

They are wrapped under a single unit Cola Vending Machine. This is called **Encapsulation**.

Abstraction

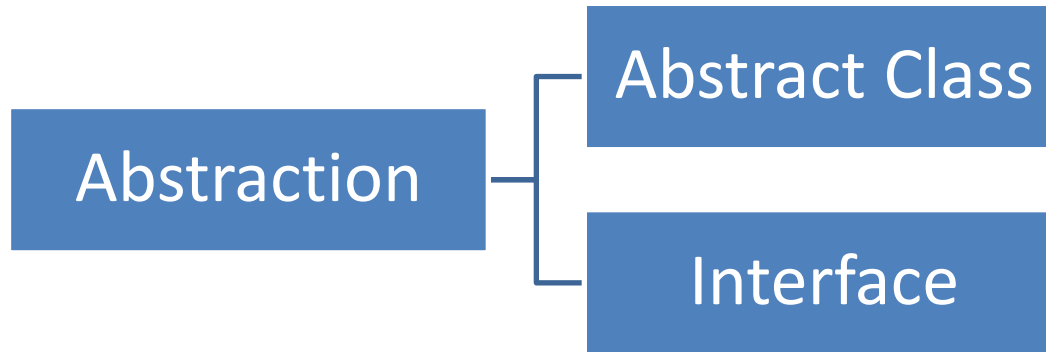
- Abstraction is the concept of exposing only the required essential characteristics and behavior with respect to a context.
- Real life example of Abstraction is ATM Machine.
- All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement...etc. but the internal details about ATM are unknown



Real Life Example of Abstraction

Abstraction

There are two ways to achieve abstraction in java



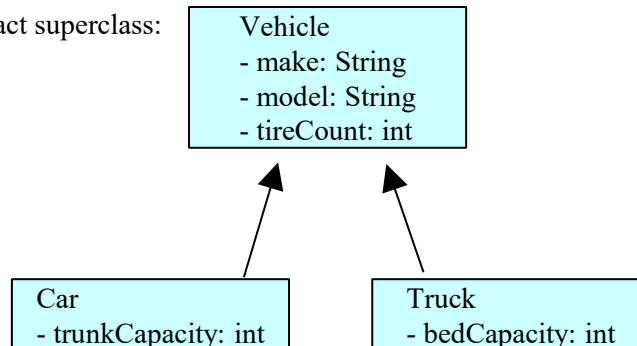
What is an Abstract class?

- **Super classes are created through the process called “generalization”.**
 - Common features (methods or variables) are factored out of object classifications (i.e. classes).
 - Those features are formalized in a class. This becomes the superclass
 - The classes from which the common features were taken become subclasses to the newly created super class
- **Often, the superclass does not have a "meaning" or does not directly relate to a "thing" in the real world**
 - It is an artifact of the generalization process

Abstract Class Example

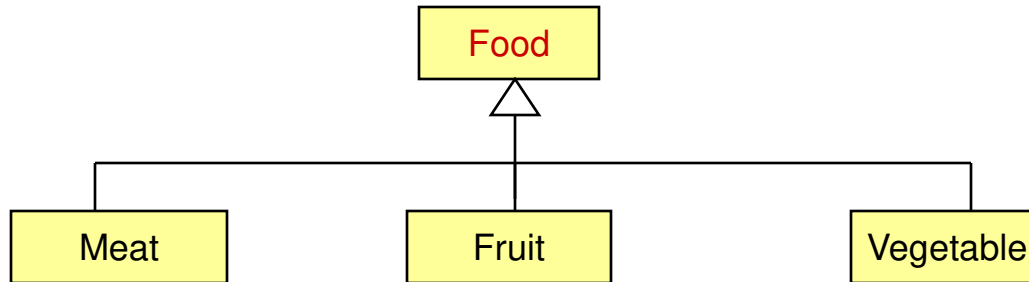
- In the following example, the subclasses represent objects taken from the problem domain.
- The superclass represents an abstract concept that does not exist "as is" in the real world

Abstract superclass:

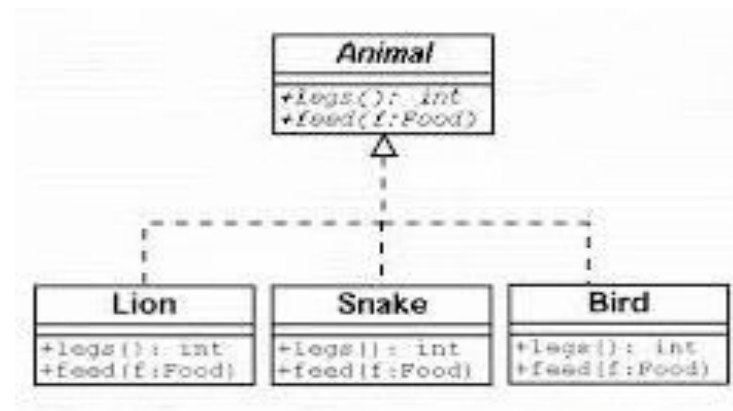
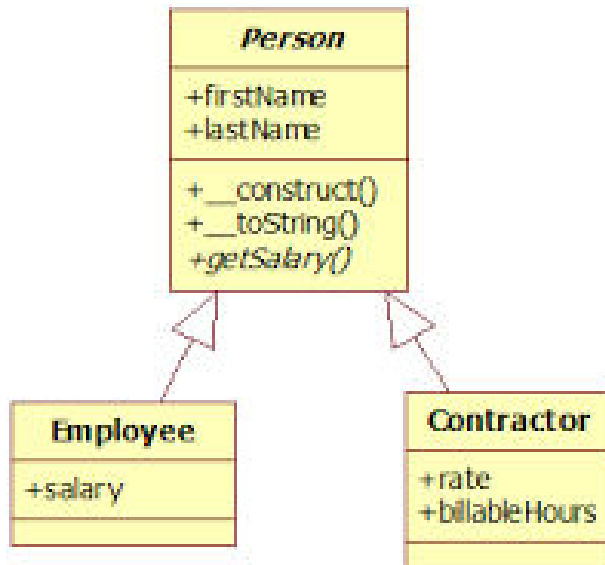
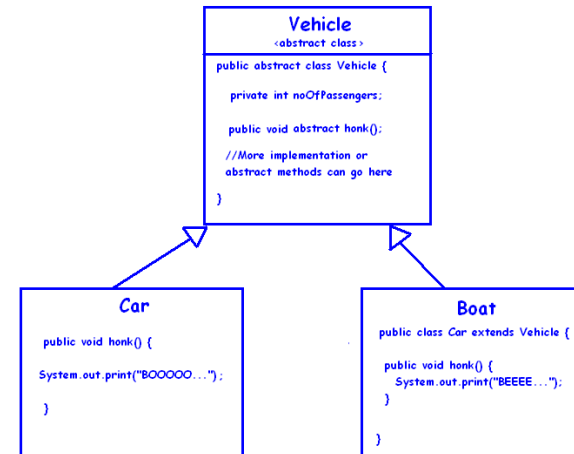
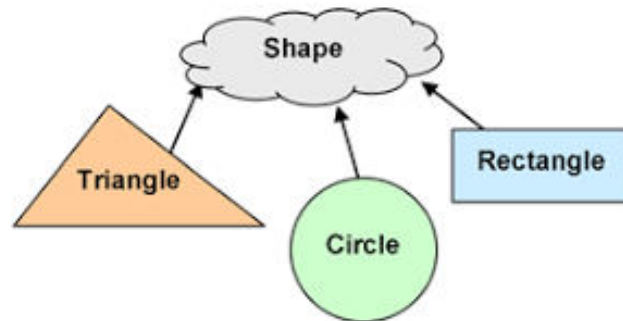


Abstract Classes

- Java allows **abstract** classes
 - use the modifier abstract on a class header to declare an abstract class
abstract class Vehicle
{ ... }
- An abstract class is a placeholder in a class hierarchy that represents a generic concept



Examples



Why have abstract classes?

- Suppose you wanted to create a class **Shape**, with subclasses **Oval**, **Rectangle**, **Triangle**, **Hexagon**, etc.
- You don't want to allow creation of a "Shape"
 - Only particular shapes make sense, not generic ones
 - If **Shape** is abstract, you can't create a **new Shape**
 - You can create a **new Oval**, a **new Rectangle**, etc.
- Abstract classes are good for defining a general category containing specific, "concrete" classes.

An example abstract class

```
public abstract class Animal {  
    abstract int eat();  
    abstract void breathe();  
}
```

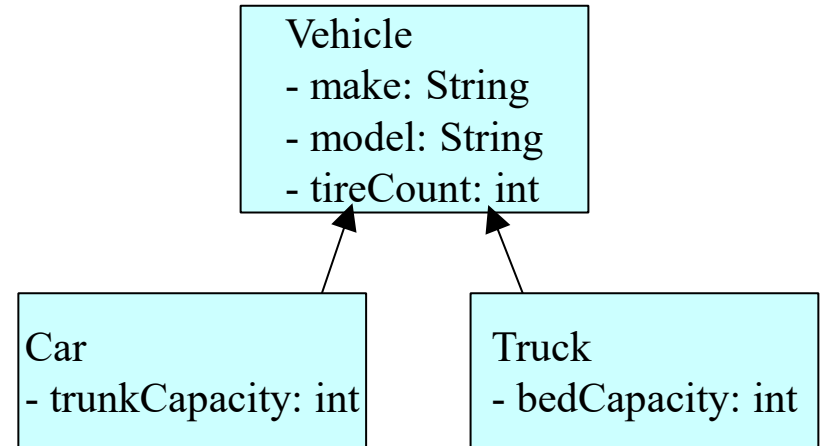
- This class cannot be instantiated
- Any non-abstract subclass of Animal must provide the `eat()` and `breathe()` methods

Defining Abstract Classes

```
public abstract class Vehicle  
{  
    private String make;  
    private String model;  
    private int tireCount;  
    [...]
```

```
public class Car extends Vehicle  
{  
    private int trunkCapacity;  
    [...]
```

```
public class Truck extends Vehicle  
{  
    private int bedCapacity;  
    [...]
```



Abstract Methods

- **Methods can also be abstracted**
 - An abstract method is one to which a signature has been provided, but no implementation for that method is given.
 - An Abstract method is a placeholder. It means that we declare that a method must exist, but there is no meaningful implementation for that methods within this class
- **Any class which contains an abstract method MUST also be abstract**
 - Any class which has an incomplete method definition cannot be instantiated (ie. it is abstract)
- **Abstract classes can contain both concrete and abstract methods.**
 - If a method can be implemented within an abstract class, and implementation should be provided.

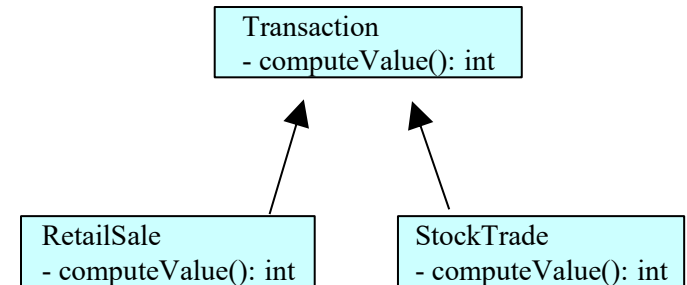
Defining Abstract Methods

```
public abstract class Transaction
{
    public abstract int
    computeValue();
}
```

```
public class RetailSale extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

```
public class StockTrade extends Transaction
{
    public int computeValue()
    {
        [...]
    }
}
```

Note: no implementation



Summary

- A method that has been declared but not defined is an **abstract method**.
- Any class containing an abstract method is an **abstract class**
- You must declare the class with the keyword **abstract**:
`abstract class MyClass {...}`
- An abstract class is *incomplete*
 - It has “missing” method bodies
- You cannot **instantiate** (create a new instance of) an abstract class

Summary

- You can extend (subclass) an abstract class
 - If the subclass defines all the inherited abstract methods, it is “complete” and can be instantiated
 - If the subclass does *not* define all the inherited abstract methods, it too must be abstract
- You can declare a class to be **abstract** even if it does not contain any abstract methods
 - This prevents the class from being instantiated

Interface

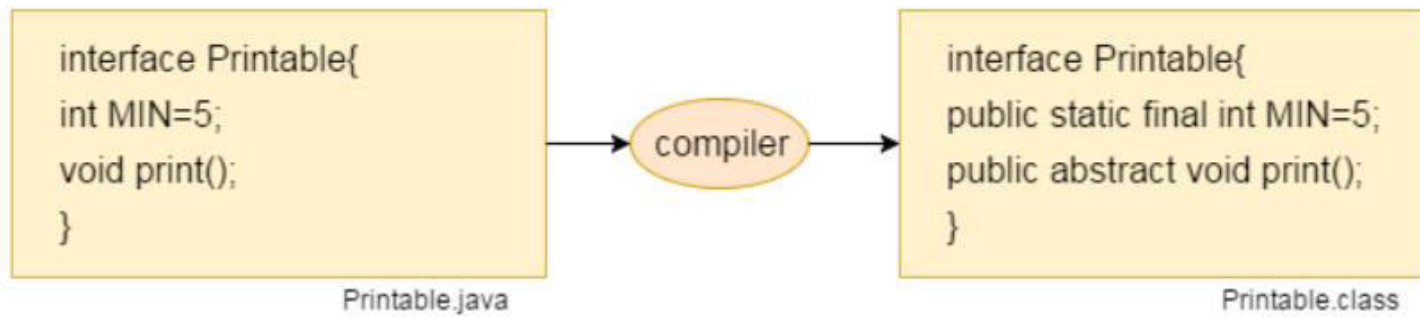
- Another way to achieve [abstraction](#) in Java, is with interfaces.
- Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).
- An interface is a completely “abstract class” that is used to group related methods with empty bodies.
- Java Interface also represents the IS-A relationship.
- To access the interface methods, the interface must be "implemented" by another class.
- Like abstract classes, interfaces cannot be used to create objects

Interfaces

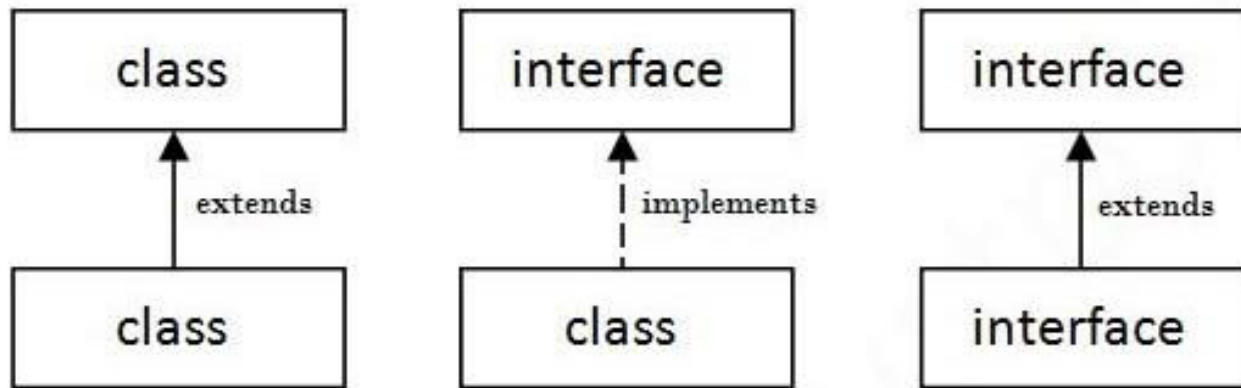
- An interface declares (describes) methods but does not provide bodies for them
 - All the methods are implicitly **public** and **abstract**
 - You cannot **instantiate** an interface
 - Interface attributes are by default **public**, **static** and **final**
 - On implementation of an interface, you must override all of its methods
-
- Since Java 8, we can have **default and static methods** in an interface.
 - Since Java 9, we can have **private methods** in an interface.

Internal addition by the compiler

- Interface fields are public, static and final by default, and
- Interface methods are public and abstract.



The relationship between classes and interfaces



Example

```
interface Animal {  
    public void animalSound(); // interface method (does not have a body)  
    public void sleep(); // interface method (does not have a body)  
}
```

```
class Pig implements Animal {  
    public void animalSound() {  
        System.out.println("The pig says: wee wee");  
    }  
    public void sleep() {  
        System.out.println("Zzz");  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Pig myPig = new Pig();  
        myPig.animalSound();  
        myPig.sleep();  
    }  
}
```

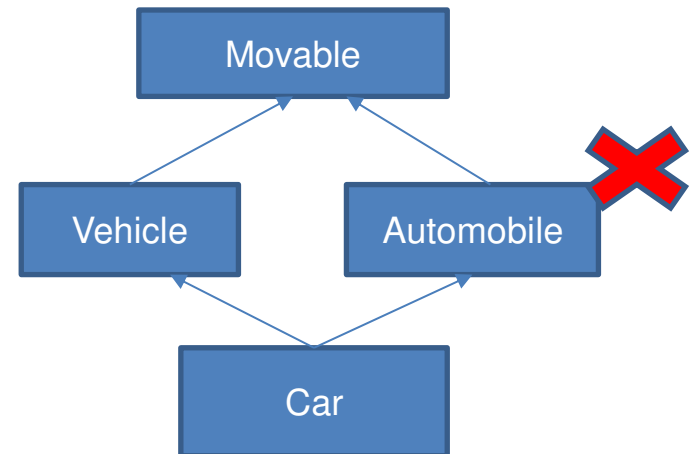
Output:

The pig says: wee wee
Zzz

Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

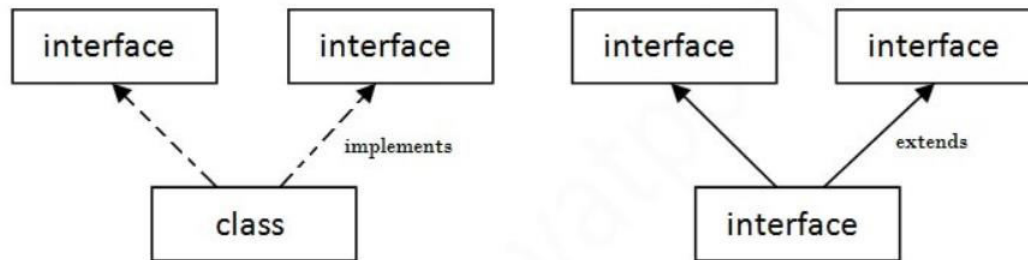
- Multiple inheritance is not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Public class Car extends Vehicle, Automobile



Multiple inheritance in Java by interface

- If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Example

```
interface Printable{  
    void print();  
}
```

```
interface Showable{  
    void show();  
}
```

```
class A7 implements Printable,Showable{  
    public void print(){System.out.println("Hello");}  
    public void show(){System.out.println("Welcome");}
```

```
public static void main(String args[]){  
    A7 obj = new A7();  
    obj.print();  
    obj.show();  
    }  
}
```

```
Output:Hello  
        welcome
```


Why And When To Use Interfaces?

1. To achieve security - hide certain details and only show the important details of an object (interface).
2. Java does not support "multiple inheritance" (a class can only inherit from one superclass). However, it can be achieved with interfaces, because the class can implement multiple interfaces.
3. It can be used to achieve loose coupling.

Abstract class	Interface
1) Abstract class can have abstract and non-abstract methods.	Interface can have only abstract methods. Since Java 8, it can have default and static methods also.
2) Abstract class doesn't support multiple inheritance.	Interface supports multiple inheritance.
3) Abstract class can have final, non-final, static and non-static variables.	Interface has only static and final variables.
4) Abstract class can provide the implementation of interface.	Interface can't provide the implementation of abstract class.
5) The abstract keyword is used to declare abstract class.	The interface keyword is used to declare interface.
6) An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
7) An abstract class can be extended using keyword "extends".	An interface can be implemented using keyword "implements".
8) A Java abstract class can have class members like private, protected, etc.	Members of a Java interface are public by default.
9)Example: <pre>public abstract class Shape{ public abstract void draw(); }</pre>	Example: <pre>public interface Drawable{ void draw(); }</pre>

Syntax

- **abstract class:**

public class Apple extends Food { ... }

- **interface:**

public class Person implements Student, Athlete, Chef { ... }