<http://beginnersbook.com/2013/04/oops-concepts/>

**Programming techniques**

a) **Unstructured Programming** (Assembly language programming)  
b) **Procedural Programming** (Assembly language, C programming)  
c) **Object Oriented Programming** (C++, Java, Smalltalk, C#, Objective C)

**Unstructured Programming**

This consists of just writing the sequence of commands or statements in the main program, which modifies the state maintained in Global Data. Example: Assembly Language programs.

**Limitations of Unstructured Programming**  
a) The data is global and code operates on it  
b) As the size of code increases, maintenance is a problem  
c) Does not have independent data for processing  
d) The concept of local variables did not exist  
e) Reusability of code was not supported

**Assembly Language:** Similar to machine language, but provides names for numeric instructions present in the machine language, making it easy for the programmer.

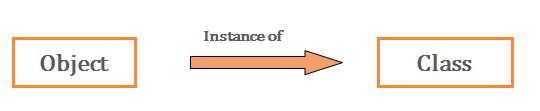
**Machine language** is the language which a Central Processing Unit (CPU) of a computer understands and consists only of numbers.

#### 2. Variables

Global Variables  
a) The variables that are declared outside any **function** body.  
b) These variables exist for the **entire life-cycle** of the program.  
c) Global variables can be accessed from anywhere within the program.

Local Variables  
a) The variables that are declared within a function body.  
b) Their scope is limited to within the function body.  
c) Local variables cannot be accessed outside the function body.

#### Object Oriented Programming



Object  
**Object:**  is a bundle of **related variables** and **functions** (also known methods).

Objects share two characteristics: They have **State and Behavior**.  
**State:** State is a well defined condition of an item. A state captures the relevant aspects of an object  
**Behavior:** Behavior is the observable effects of an operation or event,

**Examples:**  
eg 1:  
**Object**: House  
**State**: Current Location, Color, Area of House etc  
**Behavior**: Close/Open main door.

eg 2:  
**Object**: – Car  
**State**: Color, Make  
**Behavior**: Climb Uphill, Accelerate, SlowDown etc

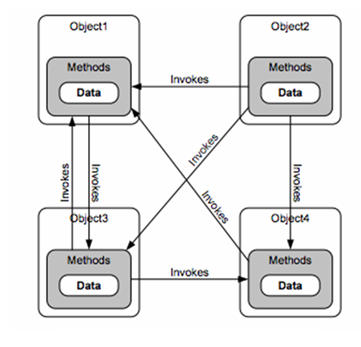
**Note:** Everything a software object knows (State) and can do (Behavior) is represented by variables and methods (functions) in the object respectively.

**Characteristics of Objects:**

1. Abstraction
2. Encapsulation
3. Message passing

#### Message passing

A single object by itself may not be very useful. An application contains many objects. One object interacts with another object by invoking methods (or functions) on that object. Through the interaction of objects, programmers achieve a higher order of functionality which has complex behavior.  
One object invoking methods on another object is known as **Message passing**.  
It is also referred to as **Method Invocation**.



#### Class

A class is a prototype that defines the variables and the methods common to all objects of a certain kind. Member Functions operate upon the member variables of the class. An Object is created when a class in instantiated.

How to create an Object?  
An object is created when a class is instantiated

Declaring an Object of class :

ClassName Objectname;

Object definition is done by calling the class constructor

Object definition is done by calling the class constructor

**Constructor**: A special member function which will be called automatically to initialize the data member of a class whenever object is instantiated.  
**Memory space** is allocated only when a class is instantiated i.e. when an object is created

**Defining a variable :**  
E.g. GraduationCourse mycourse= new GraduationCourse();

### Abstraction

The purpose of abstraction is to **hide information that is not relevant** or rather **show only relevant information** and to simplify it by comparing it to something similar in the real world.

**Abstraction means** “**The process of forming of general and relevant concept from more complex scenarios”.**  
**Note 1:** Abstraction is used to build complex systems.  
**Key to simplify a complex design into smaller, manageable parts which then become the components of a bigger and complex system**.  
The idea of hiding the complexity within a smaller/simple component of a system.  
**Note 2:** Abstraction is not a feature of Object oriented concepts alone. Even in procedural language programming, abstraction can be achieved to a limited extent by hiding complex internals through well-formed business logic and functions.

### Encapsulation

**Encapsulation** means the localization of the information or knowledge within an object.  
**Encapsulation is also called as “Information Hiding”**.  
1) Objects encapsulate data and implementation details. To the outside world, an object is a black box that exhibits a certain behavior.  
2) The behavior of this object is what which is useful for the external world or other objects.  
3) An object exposes its behavior by means of methods or functions.  
4) The set of functions an object exposes to other objects or external world acts as the interface of the object.

**Benefits of Encapsulation**  
1) The functionality where in we can change the implementation code without breaking the code of others who use our code is the biggest benefit of Encapsulation.  
2) Here in encapsulation we hide the implementation details behind a public programming interface. By interface, we mean the set of accessible methods our code makes available for other code to call—in other words, our code’s API.  
3) By hiding implementation details, We can rework on our method code at a later point of time, each time we change out implementation this should not affect the code which has a reference to our code, as our API still remains the same

**How to bring in Encapsulation**  
1) Make the instance variables protected.  
2) Create public accessor methods and use these methods from within the calling code.  
3) Use the JavaBeans naming convention of **getter and setter**.  
Eg: **getPropertyName**, **setPropertyName**.

**Example for encapsulation**

class EmployeeCount

{

private int NoOfEmployees = 0;

public void setNoOfEmployees (int count)

{

NoOfEmployees = count;

}

public double getNoOfEmployees ()

{

return NoOfEmployees;

}

}

class Encapsulation

{

public static void main(String args[])

{

System.out.println("Starting EmployeeCount...");

EmployeeCount employeeCount = new EmployeeCount ();

employeeCount. setNoOfEmployees (12003);

System.out.println("NoOfEmployees = " + employeeCount. getNoOfEmployees ());

}

}

**Takeaway from above example:**  
The application using an Object of this class EmployeeCount will not able to get the NoOfEmployees directly.  
Setting and getting the value of the field NoOfEmployees is done with the help of Getter and setter method.

### Inheritance

The process by which one class acquires the properties and functionalities of another class. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it.

1. Inheritance is a mechanism of defining a new class based on an existing class.
2. Inheritance enables reuse of code. Inheritance also provides scope for refinement of the existing class. Inheritance helps in **specialization**
3. The existing (or original) class is called the **base class** or **super class** or**parent class.** The new class which inherits from the base class is called the derived class or **sub class** or **child class**.
4. Inheritance implements the “**Is-A**” or “**Kind Of/ Has-A**” relationship.

**Note :** The biggest advantage of Inheritance is that, code in base class need not be rewritten in the derived class.  
The **member variables** and **methods** of the base class can be used in the**derived class** as well.

**Inheritance Example**  
Consider below two classes –

Class Teacher:

class Teacher {

private String name;

private double salary;

private String subject;

public Teacher (String tname)  {

name = tname;

}

public String getName()  {

return name;

}

private double getSalary()  {

return salary;

}

private String  getSubject()  {

return  subject;

}

}

Class: OfficeStaff

class  OfficeStaff{

private String name;

private double salary;

private String dept;

public OfficeStaff (String sname)  {

name = sname;

}

public String getName()  {

return name;

}

private double  getSalary()  {

return salary;

}

private String  getDept ()  {

return dept;

}

}

Points:  
1) Both the classes share few common properties and methods. Thus repetition of code.  
2) Creating a class which contains the common methods and properties.  
3) The classes Teacher and OfficeStaff can inherit the all the common properties and methods from below Employee class

class Employee{

private String name;

private double salary;

public Employee(String ename){

name=ename;

}

public String getName(){

return name;

}

private double getSalary(){

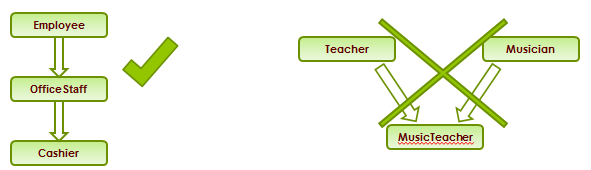
return salary;

}

}

4) Add individual methods and properties to it Once we have created a super class that defines the attributes common to a set of objects, it can be used to create any number of more specific subclasses  
5) Any similar classes like Engineer, Principal can be generated as subclasses from the Employee class.  
6) The parent class is termed super class and the inherited class is the sub class  
7) A sub class is the specialized version of a super class – It inherits all of the instance variables and methods defined by the super class and adds its own, unique elements.  
8) Although a sub class includes all of the members of its super class it can not access those members of the super class that have been declared as private.  
9) A reference variable of a super class can be assigned to a reference to any sub class derived from that super class  
i.e. Employee emp = new Teacher();

**Note: Multi-level inheritance** is allowed in Java but **not multiple inheritance**



**Types of Inheritance**  
**Multilevel Inheritance**  
**Multilevel inheritance** refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.  
**Multiple Inheritance**  
“**Multiple Inheritance**” refers to the concept of one class inheriting from more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.  
**Note 1**: **Multiple Inheritance** is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.  
**Note 2:** Most of the new **OO** **languages** like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

### Polymorphism

**Polymorphism** is a feature that allows one interface to be used for a general class of actions. It’s an operation may exhibit different behavior in different instances. The behavior depends on the types of data used in the operation. It plays an important role in allowing objects having different internal structures to share the same external interface. Polymorphism is extensively used in implementing inheritance.

**Types of Polymorphism**  
1) Static Polymorphism  
2) Dynamic Polymorphism

**Static Polymorphism:**  
**Function Overloading** – within same class more than one method having same name but differing in signature.  
Resolved during compilation time.  
Return type is not part of method signature.

**Dynamic Polymorphism**  
**Function Overriding** – keeping the signature and return type same, method in the base class is redefined in the derived class.  
Resolved during run time.  
Which method to be invoked is decided by the object that the reference points to and not by the type of the reference.

**Overriding:**  
Redefining a super class method in a sub class is called method overriding.  
The method signature ie. method name, parameter list and return type have to match exactly.  
The overridden method can widen the accessibility but not narrow it, ie if it is private in the base class, the child class can make it public but not vice versa.

**Overriding Examples:**  
Consider a Super Class Doctor and Subclass Surgeon. Class Doctor has a method treatPatient(). Surgeon overrides treatPatient method ie gives a new definition to the method.

Doctor doctorObj = new Doctor();

// Call the treatPatient method of Doctor class

doctorObj.treatPatient()

Surgeon surgeonObj = new Surgeon();

// Call the treatPatient method of Surgeon class

surgeonObj.treatPatient()

Doctor obj = new Surgeon();

// calls Surgeon’s treatPatient method as the reference is pointing to Surgeon

obj.treatPatient();

**Method/Function Overloading:**  
Method Overloading refers to the practice of using the same name to denote several different operations. Overloading can be done for both functions as well as operators. Here we look at only Method overloading.  
**Declaration:**

void SomeMethod (int value);

void SomeMethod (float value);

void SomeMethod (char value);

void SomeMethod (String\* str);

void SomeMethod (char\* str);

All the five methods are called ‘SomeMethod ’. All the methods have the same name, but different signatures.

**The concept of the same function name with different types of parameters being passed is called Function Overloading.**  
1) In Overloading we can reuse the same method name by changing the arguments.  
2) Overloaded methods- Must and Must Not Facts:

* The Overloaded method must have different argument lists,
* Can have different return types but in that case it is mandatory to have different argument list.
* Can have different access modifiers and
* Can throw different exceptions

3) Methods can be overloaded in the same as well as the sub classes.

**Q: What determines which overridden method is used at runtime?  
A: Object type  
Q: What determines which overloaded method will be used at compile time?  
A: Reference type determines.**Operator overloading refers to the operators like ‘+’ being used for different purposes based on the data type on either side of the operator.

#### Combined example for Inheritance & Polymorphism

abstract public class Employee {

private String name;

public Employee(String ename) {

name = ename;

}

public String getName() {

return new name;

}

private void setName(String name) {

this.name = new String(name);

}

abstract public double pay();

public String toString() {

return "name is" + name;

}

}

public class Salaried extends Employee {

double salary;

public Salaried(String name, double s) {

super(name);

salary =s;

}

public void setSalary(double salary) {

this.salary = salary;

}

public double getSalary() {

return salary;

}

public double pay() {

return salary;

}

public String toString(){

return super.toString() + " (salary is " +salary+")";

}

}

#### IS-A & HAS-A Relationships

public class SuperClass { … }

public class SubClass1 extends SuperClass { //SubClass 1 code goes here }

public class SubClass2 extends SubClass1 { //SubClass2 Specific code goes here }

**HAS-A** relationships are based on usage, rather than inheritance. In other words, class A **HAS-A** B if code in class A has a reference to an instance of class B.

For example, we can say the following,  
A Car **IS-A** Vehicle. A Car **HAS-A** License. and the code looks like this:

public class Vehicle{ }

public class Car extends Vehicle{

private License myCarLicense;

}

### Interfaces

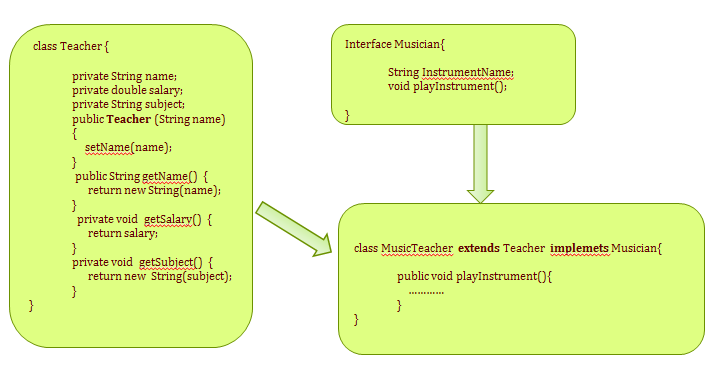
As in the example Teacher class contains the attributes and methods of Employee as well as Musician.

1. Java does not support Multiple Inheritance
2. Interface is similar to an abstract class that contains only abstract methods
3. Declared with the keyword interface instead of the keyword class
4. Keyword implements is used to represent the interfaces implemented by the class

**Interface: Syntax**

class ClassName extends Superclass implements Interface1, Interface2, ....

**Example :** class MusicTeacher extends Teacher implements Musician



*All****methods in an interface****are implicitly public and abstract.  
The keyword****abstract****before each method is optional.  
An****interface****may contain definitions of final variables.  
A class may****extend only one other class****, but it may* ***implement any number of interfaces.*** *When a class implements an interface, it may implement (define) some or all of the****inherited abstract methods.*** *A class must itself be declared abstract if it inherits* ***abstract methods****that it does not define.  
An interface reference can point to****objects****of its implementing classes.*

**Abstract method**  
1) A method that is declared but not defined  
2) Declared with the keyword abstract  
3) Example :

abstract public void playInstrument();

4) Has a header like any other method, but ends with a semicolon instead of a method body  
5) Used to put some kind of compulsion on the class who inherits from this class. i.e., the class who inherits MUST provide the implementation of the method else the subclass will also become abstract  
6) The following cannot be marked with “abstract” modifier

* Constructors
* Static methods
* Private methods
* Methods marked with “final” modifier

### Abstract Classes

**Abstract Classes**  
Outlines the behavior but not necessarily implements all of its behavior. Also known as **Abstract Base Class**.  
Provides outline for behavior by means of method (abstract methods) **signatures** without an implementation.

**Note 1:** There can be some scenarios where it is difficult to implement all the methods in the base class. In such scenarios one can define the base class as an abstract class which signifies that this base class is a special kind of class which is not complete on its own.  
A class derived from the abstract base class must implement those member functions which are not implemented in the abstract class.

**Note 2:** Abstract Class cannot be instantiated.  
To use an abstract class one has to first derive another class from this abstract class using inheritance and then provide the implementation for the abstract methods.  
**Note 3:** If a derived class does not implement all the abstract methods (unimplemented methods), then the derived class is also abstract in nature and cannot be instantiated.

**Example of Abstract class and Abstract Method:**

abstract class Costume {

abstract public void Stitch();

public void  setColour (){

//code to set colour

}

}

Here the class which inherits abstract class Costume can implement the abstract method Stitch depending upon what kind of Costume it is.

### What is a Constructor

1) A method with the same name as class name used for the purpose of creating an object in a valid state  
2) It does not return a value not even void  
3) It may or may not have parameters (arguments)  
4) A class contains one or more constructors for making new objects of that class  
5) If (and only if) the programmer does not write a constructor, Java provides a default constructor with no arguments.

**The default constructor sets instance variables as:**

* numeric types are set to zero
* boolean variables are set to false
* char variables are set to ‘’
* object variables are set to null.

**When a constructor executes, before executing its own code:**  
It implicitly call the default constructor of it’s super class Or can make this constructor call explicitly, with super(…);

A constructor for a class can call another constructor for the same class by putting this(…); as the first thing in the constructor. This allows you to avoid repeating code.

**A Class do not inherit the constructors of a super class.**

**Generalization and Specialization:**  
In order to implement the concept of inheritance in an OO solution, one has to first identify the similarities among different classes so as to come up with the base class.

This process of identifying the similarities among different classes is called**Generalization**.

The class which is identified after generalization is the base class. The classes over which the generalization is made are the derived classes.

The classes over which the generalization is built are viewed as**Specialization**.

**Generalization** identifies the common attributes and behavior among different entities whereas specialization identifies the distinct attributes and their behavior which differentiate a derived class from the other classes.

A **Generalization–Specialization hierarchy** is used to depict the relationship between the generalized class and the corresponding specialized classes.

#### Access Specifiers

**Access specifiers in a class**  
An access specifier is a keyword in **OO Programming languages**, which specify what access is permitted on a member.  
There are **three types** of access specifiers:  
**public:** Accessible to all. Other objects can also access this member variable or function.  
**private:** Not accessible by other objects. Private members can be accessed only by the methods in the same class. **Object accessible only in Class in which they are declared.**  
**protected:** The scope of a protected variable is within the class which declares it and in the class which inherits from the class (Scope is Class and subclass)  
**Default:** Scope is Package Level

**Approach to Object Oriented Design:**  
**Step 1**. Identify all the classes (Nouns; Type of objects) in the requirement specification,  
**Step 2**. Identify the commonalities between all or small groups of classes identified (generalization) if it is obvious. Do not force fit generalization where it doesn’t make sense.  
**Step 3**. In any given situation, start with the simplest object which can be abstracted into individual classes  
**Step 4**. Identify all the member variables and methods the class should have  
**Step 5**. Ensure that the class is fully independent of other classes and contains all the attributes and methods necessary.  
**Step 6**. Keep all the data members private or protected  
**Step 7**. The methods in the class should completely abstract the functionality  
**Step 8**. The methods in the class should not be a force fit of procedural code into a class  
**Step 9**. Inherit and extend classes from the base classes ONLY IF the situation has scope for it  
**Step 10**. Define the relationships among the classes (“Has-A”, “Uses-A”)  
**Step 11**. Keep the number of classes in your application under check (do not create any unnecessary classes)