OOPS

Inheritance

**When one object acquires all the properties and behaviours of parent object** i.e. known as inheritance.

**It provides code reusability.**

**It is used to achieve runtime polymorphism.**

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.**

1. 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**.
2. **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;

}

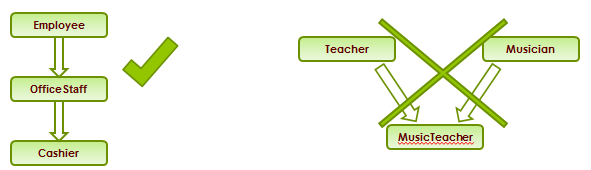
}

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.->private is only meant for that class and protected child classes can implement**

**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**

When **one task is performed by different ways** i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In java, we use method overloading and method overriding to achieve polymorphism.

Another example can be to speak something e.g. cat speaks meaw, dog barks woof etc.

**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 Doctorhas 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?

Which method to be invoked is decided by the object that the reference points to and not by the type of the reference.

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;

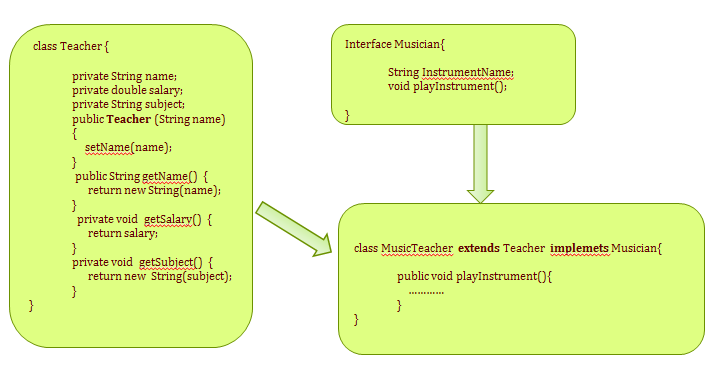
}

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.

### 6.18. Constructors

* Constructors are like special methods that are called implicitly as soon as an object is instantiated (i.e. on **new ClassName()**).
  + Constructors have no return type (not even **void**).
  + The constructor name must match the class name.
* If you don’t define an explicit constructor, Java assumes a default constructor
  + The default constructor accepts no arguments.
  + The default constructor automatically invokes its base class constructor with no arguments, as discussed later in this module.
* You can provide one or more explicit constructors to:
  + Simplify object initialization (one line of code to create and initialize the object)
  + Enforce the state of objects (require parameters in the constructor)
  + Invoke the base class constructor with arguments, as discussed later in this module.
* Adding any explicit constructor disables the implicit (no argument) constructor.

We can add a constructor to our Employee class to allow/require that it be constructed with a name and a social security number:

class Employee {

String name;

String ssn;

…

Employee(String name, String ssn) {

this.name = name; // "this." helps distinguish between

this.ssn = ssn; // instance and parameter variables

}

…

}

Then we can modify **EmployeeDemo.main()** to call the specified constructor:

public class EmployeeDemo {

public static void main(String[] args) {

Employee e1 = new Employee("John", "555-12-345");

e1.emailAddress = "john@company.com";

Employee e2 = new Employee("Tom", "456-78-901");

e2.setYearOfBirth(1974);

…

}

}

### 6.19. Constructors (cont.)

* **As with methods, constructors can be overloaded.**
* **Each constructor must have a unique signature.**
  + **The parameter type list must be different, either different number or different order.**
  + **Only parameter types determine the signature, not parameter names.**
* One constructor can invoke another by invoking **this(*param1, param2, …*)** as the first line of its implementation.

It is no longer possible to do **Employee e = new Employee();** because there is no constructor that takes no parameters.

We could add additional constructors to our class Employee:

Employee(String ssn) { // employees must have at least a SSN

this.ssn = ssn;

}

Employee(String name, String ssn) {

this(ssn);

this.name = name;

}

Employee(String name, String ssn, String emailAddress) {

this(name, ssn);

this.emailAddress = emailAddress;

}

Employee(String ssn, int yearOfBirth) {

this(ssn);

this.yearOfBirth = yearOfBirth;

}

Now we can construct Employee objects in different ways:

Employee e1 = new Employee("John", "555-12-345", "john@company.com");

Employee e2 = new Employee("456-78-901", 1974);

e2.name = "Tom";

### 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)

### 6.16. Method Overloading

* A class can provide multiple definitions of the same method. This is known as overloading.
* Overloaded methods must have distinct signatures:
  + **The parameter type list must be different, either different number or different order.**
  + **Only parameter types determine the signature, not parameter names.**
  + **The return type is not considered part of the signature.**

We can overload the print method in our Employee class to support providing header and footer to be printed:

public class Employee {

…

public void print(String header, String footer) {

if (header != null) {

System.out.println(header);

}

System.out.println("Name: " + name);

System.out.println("SSN: " + ssn);

System.out.println("Email Address: " + emailAddress);

System.out.println("Year Of Birth: " + yearOfBirth);

System.out.println("Vacation Days: " + getVacationDays());

if (footer != null) {

System.out.println(footer);

}

}

public void print(String header) {

print(header, null);

}

public void print() {

print(null);

}

}

In our **EmployeeDemo.main()** we can then do:

e1.print("COOL EMPLOYEE");

e2.print("START OF EMPLOYEE", "END OF EMPLOYEE");

Abstraction

**Hiding internal details and showing functionality** is known as abstraction. For example: phone call, we don't know the internal processing.

**In java, we use abstract class and interface to achieve 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

**Binding (or wrapping) code and data together into a single unit is known as encapsulation**. For example: capsule, it is wrapped with different medicines.

A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.

**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 as shown below.

#### public vs. private - Access Control Modifiers

An access control modifier can be used to control the visibility of a class, or a member variable or a member method within a class. We begin with the following two access control modifiers:

1. **public: The class/variable/method is accessible and available to all the other objects in the system.**
2. **private: The class/variable/method is accessible and available within this class only.**

For example, in the above Circle definition, the member variable radius is declared private. As the result, radius is accessible inside the Circle class, but NOT in the TestCircle class. In other words, you cannot use "c1.radius" to refer to c1's radius in TestCircle.

* Try inserting the statement "System.out.println(c1.radius)" in TestCircle and observe the error message.
* Try changing radius to public in the Circle class, and re-run the above statement.

On the other hand, the method getRadius() is declared public in the Circle class. Hence, it can be invoked in the TestCircle class.

**UML Notation:** In UML class diagram, public members are denoted with a "+"; while private members with a "-".

More access control modifiers will be discussed later.

#### 2.12  Information Hiding and Encapsulation

**A class encapsulates the name, static attributes and dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelve for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.**

Member variables of a class are typically hidden from the outside word (i.e., the other classes), with private access control modifier. Access to the member variables are provided via public assessor methods, e.g., getRadius() and getColor().

This follows the principle of information hiding. That is, objects communicate with each others using well-defined interfaces (public methods). Objects are not allowed to know the implementation details of others. The implementation details are hidden or encapsulated within the class. Information hiding facilitates reuse of the class.

**Rule of Thumb:** Do not make any variable public, unless you have a good reason.

#### 2.13  The public Getters and Setters for private Variables

To allow other classes to read the value of a private variable says xxx, we provide a get method (or getter or accessor method) called getXxx(). A get method needs not expose the data in raw format. It can process the data and limit the view of the data others will see. The getters shall not modify the variable.

To allow other classes to modify the value of a private variable says xxx, we provide a set method (or setter or mutator method) called setXxx(). A set method could provide data validation (such as range checking), or transform the raw data into the internal representation.

For example, in our Circle class, the variables radius and color are declared private. That is to say, they are only accessible within the Circle class and not visible in any other classes, such as the TestCircle class. You cannot access the private variables radius and color from the TestCircle class directly - via says c1.radius or c1.color. The Circle class provides two public accessor methods, namely, getRadius() and getColor(). These methods are declared public. The class TestCircle can invoke these public accessor methods to retrieve the radius and color of a Circle object, via says c1.getRadius() and c1.getColor().

There is no way you can change the radius or color of a Circle object, after it is constructed in the TestCircle class. You cannot issue statements such as c1.radius = 5.0 to change the radius of instance c1, as radius is declared as private in the Circle class and is not visible to other classes including TestCircle.

If the designer of the Circle class permits the change the radius and color after a Circle object is constructed, he has to provide the appropriate set methods (or setters or mutator methods), e.g.,

// Setter for color

public void setColor(String newColor) {

color = newColor;

}

// Setter for radius

public void setRadius(double newRadius) {

radius = newRadius;

}

With proper implementation of information hiding, the designer of a class has full control of what the user of the class can and cannot do.

#### 2.14  Keyword "this"

You can use keyword "this" to refer to thisinstance inside a class definition.

One of the main usage of keyword this is to resolve ambiguity.

public class Circle {

double radius; // Member variable called "radius"

public Circle(double radius) { // Method's argument also called "radius"

this.radius = radius;

// "radius = radius" does not make sense!

// "this.radius" refers to this instance's member variable

// "radius" resolved to the method's argument.

}

...

}

In the above codes, there are two identifiers called radius - a member variable of the class and the method's argument. This causes naming conflict. To avoid the naming conflict, you could name the method's argument r instead of radius. However, radius is more approximate and meaningful in this context. Java provides a keyword called this to resolve this naming conflict. "this.radius" refers to the member variable; while "radius" resolves to the method's argument.

Using the keyword "this", the constructor, getter and setter methods for a private variable called xxx of type T are as follows:

public class Aaa {

// A private variable named xxx of the type T

private T xxx;

// Constructor

public Aaa(T xxx) {

this.xxx = xxx;

}

// A getter for variable xxx of type T receives no argument and return a value of type T

public T getXxx() {

return xxx; // or "return this.xxx" for clarity

}

// A setter for variable xxx of type T receives a parameter of type T and return void

public void setXxx(T xxx) {

this.xxx = xxx;

}

}

For a boolean variable xxx, the getter shall be named isXxx() or hasXxx(), which is more meaningful than getXxx(). The setter remains setXxx().

// Private boolean variable

private boolean xxx;

// Getter

public boolean isXxx() {

return xxx; // or "return this.xxx" for clarity

}

// Setter

public void setXxx(boolean xxx) {

this.xxx = xxx;

}

##### More on "this"

* this.varName refers to *varName* of this instance; this.methodName(...) invokes methodName(...) of this instance.
* In a constructor, we can use this(...) to call another constructor of this class.
* Inside a method, we can use the statement "return this" to return this instance to the caller.

#### 2.15  Method toString()

Every well-designed Java class should have a public method called toString() that returns a string description of this instance. You can invoke the toString() method explicitly by calling anInstanceName.toString(), or implicitly via println() or String concatenation operator '+'. **That is, running println(anInstance) invokes the toString() method of that instance implicitly.**

For example, include the following toString() method in our Circle class:

// Return a String description of this instance

public String toString() {

return "Circle[radius=" + radius + ",color=" + color + "]";

}

In your TestCircle class, you can get a description of a Circle instance via:

Circle c1 = new Circle();

System.out.println(c1.toString()); // Explicitly calling toString()

System.out.println(c1); // Implicit call to c1.toString()

System.out.println("c1 is: " + c1); // '+' invokes toString() to get a String before concatenation

The signature of toString() is:

public String toString() { ...... }

#### 2.16  Constants (final)

Constants are variables defined with the modifier final. A final variable can only be assigned once and its value cannot be modified once assigned. For example,

public final double X\_REFERENCE = 1.234;

private final int MAX\_ID = 9999;

MAX\_ID = 10000; // error: cannot assign a value to final variable MAX\_ID

// You need to initialize a final member variable during declaration

private final int SIZE; // error: variable SIZE might not have been initialized

**Constant Naming Convention:** A constant name is a noun, or a noun phrase made up of several words. All words are in uppercase separated by underscores '\_', for examples, X\_REFERENCE, MAX\_INTEGER and MIN\_VALUE.

**Advanced Notes:**

1. **A final primitive variable cannot be re-assigned a new value.**
2. **A final instance cannot be re-assigned a new object.**
3. **A final class cannot be sub-classed (or extended).**
4. **A final method cannot be overridden.**

**COMPOSITION**

**O**ne of the fundamental activities of any software system design is establishing relationships between classes. Two fundamental ways to relate classes are *inheritance* and *composition*. Although the compiler and Java virtual machine (JVM) will do a lot of work for you when you use inheritance, you can also get at the functionality of inheritance when you use composition. This article will compare these two approaches to relating classes and will provide guidelines on their use.

First, some background on the meaning of inheritance and composition.

**About inheritance**  
In this article, I'll be talking about single inheritance through class extension, as in:

class Fruit {

//...

}

class Apple extends Fruit {

//...

}

In this simple example, class Apple is related to class Fruit by inheritance, because Apple extends Fruit. In this example, Fruit is the *superclass* and Apple is the *subclass*.

Here's a UML diagram showing the inheritance relationship between Apple and Fruit:

|  |
| --- |
| Inheritance relationship  **Figure 1. The inheritance relationship** |

**About composition**  
By composition, I simply mean using instance variables that are references to other objects. For example:

class Fruit {

//...

}

class Apple {

private Fruit fruit = new Fruit();

//...

}

**In the example above, class Apple is related to class Fruit by composition, because Apple has an instance variable that holds a reference to a Fruit object.** **In this example, Apple is what I will call the *front-end class* and Fruit is what I will call the *back-end class*. In a composition relationship, the front-end class holds a reference in one of its instance variables to a back-end class.**

**The UML diagram showing the composition relationship has a darkened diamond, as in:**

|  |
| --- |
| Composition relationship  **Figure 2. The composition relationship** |

**Dynamic binding, polymorphism, and change**  
When you establish an inheritance relationship between two classes, you get to take advantage of *dynamic binding* and *polymorphism.*

**Dynamic binding means the JVM will decide at runtime which method implementation to invoke based on the class of the object.**

Polymorphism means you can use a variable of a superclass type to hold a reference to an object whose class is the superclass or any of its subclasses.

One of the prime benefits of dynamic binding and polymorphism is that they can help make code easier to change.

If you have a fragment of code that uses a variable of a superclass type, such as Fruit, you could later create a brand new subclass, such as Banana, and the old code fragment will work without change with instances of the new subclass. If Banana overrides any of Fruit's methods that are invoked by the code fragment, dynamic binding will ensure that Banana's implementation of those methods gets executed. This will be true even though class Banana didn't exist when the code fragment was written and compiled.

Thus, inheritance helps make code easier to change if the needed change involves adding a new subclass. This, however, is not the only kind of change you may need to make.

**Changing the superclass interface**  
In an inheritance relationship, superclasses are often said to be "fragile," because one little change to a superclass can ripple out and require changes in many other places in the application's code. To be more specific, what is actually fragile about a superclass is its interface. If the superclass is well-designed, with a clean separation of interface and implementation in the object-oriented style, any changes to the superclass's implementation shouldn't ripple at all. Changes to the superclass's interface, however, can ripple out and break any code that uses the superclass or any of its subclasses. **What's more, a change in the superclass interface can break the code that defines any of its subclasses.**

For example, if you change the return type of a public method in class Fruit (a part of Fruit's interface), you can break the code that invokes that method on any reference of typeFruit or any subclass of Fruit. In addition, you break the code that defines any subclass of Fruit that overrides the method. Such subclasses won't compile until you go and change the return value of the overridden method to match the changed method in superclass Fruit.

Inheritance is also sometimes said to provide "weak encapsulation," because if you have code that directly uses a subclass, such as Apple, that code can be broken by changes to a superclass, such as Fruit. One of the ways to look at inheritance is that it allows subclass code to *reuse* superclass code. For example, if Apple doesn't override a method defined in its superclass Fruit, Apple is in a sense reusing Fruit's implementation of the method. But Apple only "weakly encapsulates" the Fruit code it is reusing, because changes to Fruit's interface can break code that directly uses Apple.

**The composition alternative**  
Given that the inheritance relationship makes it hard to change the interface of a superclass, it is worth looking at an alternative approach provided by composition. It turns out that when your goal is code reuse, composition provides an approach that yields easier-to-change code.

**Code reuse via inheritance**  
For an illustration of how inheritance compares to composition in the code reuse department, consider this very simple example:

class Fruit {

// Return int number of pieces of peel that

// resulted from the peeling activity.

public int peel() {

System.out.println("Peeling is appealing.");

return 1;

}

}

class Apple extends Fruit {

}

class Example1 {

public static void main(String[] args) {

Apple apple = new Apple();

int pieces = apple.peel();

}

}

When you run the Example1 application, it will print out "Peeling is appealing.", because Apple inherits (reuses) Fruit's implementation of peel(). **If at some point in the future, however, you wish to change the return value of peel() to type Peel, you will break the code for Example1**. Your change to Fruit breaks Example1's code even though Example1 uses Apple directly and never explicitly mentions Fruit.

Here's what that would look like:

class Peel {

private int peelCount;

public Peel(int peelCount) {

this.peelCount = peelCount;

}

public int getPeelCount() {

return peelCount;

}

//...

}

class Fruit {

// Return a Peel object that

// results from the peeling activity.

public Peel peel() {

System.out.println("Peeling is appealing.");

return new Peel(1);

}

}

// Apple still compiles and works fine

class Apple extends Fruit {

}

// This old implementation of Example1

// is broken and won't compile.

class Example1 {

public static void main(String[] args) {

Apple apple = new Apple();

int pieces = apple.peel();

}

}

**Code reuse via composition**  
Composition provides an alternative way for Apple to reuse Fruit's implementation of peel(). Instead of extending Fruit, Apple can hold a reference to a Fruit instance and define its own peel() method that simply invokes peel() on the Fruit. Here's the code:

class Fruit {

// Return int number of pieces of peel that

// resulted from the peeling activity.

public int peel() {

System.out.println("Peeling is appealing.");

return 1;

}

}

class Apple {

private Fruit fruit = new Fruit();

public int peel() {

return fruit.peel();

}

}

class Example2 {

public static void main(String[] args) {

Apple apple = new Apple();

int pieces = apple.peel();

}

}

In the composition approach, the subclass becomes the "front-end class," and the superclass becomes the "back-end class." With inheritance, a subclass automatically inherits an implemenation of any non-private superclass method that it doesn't override. **With composition, by contrast, the front-end class must explicitly invoke a corresponding method in the back-end class from its own implementation of the method. This explicit call is sometimes called "forwarding" or "delegating" the method invocation to the back-end object.**

The composition approach to code reuse provides stronger encapsulation than inheritance, because a change to a back-end class needn't break any code that relies only on the front-end class. For example, changing the return type of Fruit's peel() method from the previous example doesn't force a change in Apple's interface and therefore needn't break Example2's code.

Here's how the changed code would look:

class Peel {

private int peelCount;

public Peel(int peelCount) {

this.peelCount = peelCount;

}

public int getPeelCount() {

return peelCount;

}

//...

}

class Fruit {

// Return int number of pieces of peel that

// resulted from the peeling activity.

public Peel peel() {

System.out.println("Peeling is appealing.");

return new Peel(1);

}

}

// Apple must be changed to accomodate

// the change to Fruit

class Apple {

private Fruit fruit = new Fruit();

public int peel() {

Peel peel = fruit.peel();

return peel.getPeelCount();

}

}

// This old implementation of Example2

// still works fine.

class Example1 {

public static void main(String[] args) {

Apple apple = new Apple();

int pieces = apple.peel();

}

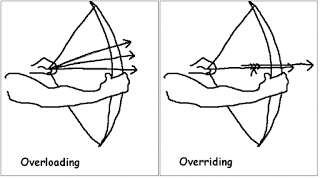
}

This example illustrates that the ripple effect caused by changing a back-end class stops (or at least can stop) at the front-end class. Although Apple's peel() method had to be updated to accommodate the change to Fruit, Example2 required no changes.

* It is easier to change the interface of a back-end class (composition) than a superclass (inheritance). As the previous example illustrated, a change to the interface of a back-end class necessitates a change to the front-end class implementation, but not necessarily the front-end interface. Code that depends only on the front-end interface still works, so long as the front-end interface remains the same. By contrast, a change to a superclass's interface can not only ripple down the inheritance hierarchy to subclasses, but can also ripple out to code that uses just the subclass's interface.
* It is easier to change the interface of a front-end class (composition) than a subclass (inheritance). Just as superclasses can be fragile, subclasses can be rigid. You can't just change a subclass's interface without making sure the subclass's new interface is compatible with that of its supertypes. For example, you can't add to a subclass a method with the same signature but a different return type as a method inherited from a superclass. Composition, on the other hand, allows you to change the interface of a front-end class without affecting back-end classes.
* Composition allows you to delay the creation of back-end objects until (and unless) they are needed, as well as changing the back-end objects dynamically throughout the lifetime of the front-end object. With inheritance, you get the image of the superclass in your subclass object image as soon as the subclass is created, and it remains part of the subclass object throughout the lifetime of the subclass.
* It is easier to add new subclasses (inheritance) than it is to add new front-end classes (composition), because inheritance comes with polymorphism. If you have a bit of code that relies only on a superclass interface, that code can work with a new subclass without change. This is not true of composition, unless you use composition with interfaces. Used together, composition and interfaces make a very powerful design tool. I'll talk about this approach in next month's **Design Techniques** article.
* The explicit method-invocation forwarding (or delegation) approach of composition will often have a performance cost as compared to inheritance's single invocation of an inherited superclass method implementation. I say "often" here because the performance really depends on many factors, including how the JVM optimizes the program as it executes it.
* With both composition and inheritance, changing the implementation (not the interface) of any class is easy. The ripple effect of implementation changes remain inside the same class.
* **Choosing between composition and inheritance**  
  So how do all these comparisons between composition and inheritance help you in your designs? Here are a few guidelines that reflect how I tend to select between composition and inheritance.
* **Make sure inheritance models the *is-a* relationship**  
  My main guiding philosophy is that inheritance should be used only when a subclass *is-a* superclass. In the example above, an Apple likely is-a Fruit, so I would be inclined to use inheritance.
* An important question to ask yourself when you think you have an is-a relationship is whether that is-a relationship will be constant throughout the lifetime of the application and, with luck, the lifecycle of the code. For example, you might think that an Employee is-a Person, when really Employee represents a role that a Person plays part of the time. What if the person becomes unemployed? What if the person is both an Employee and a Supervisor? Such impermanent is-a relationships should usually be modelled with composition.
* **Don't use inheritance just to get code reuse**  
  If all you really want is to reuse code and there is no is-a relationship in sight, use composition.
* **Don't use inheritance just to get at polymorphism**  
  If all you really want is polymorphism, but there is no natural is-a relationship, use composition with interfaces. I'll be talking about this subject next month.
* Java composition is achieved by using instance variables that refers to other objects. For example, a Personhas a Job. Let’s see this with a simple code.
* package com.journaldev.composition;
* public class Job {
* private String role;
* private long salary;
* private int id;
* public String getRole() {
* return role;
* }
* public void setRole(String role) {
* this.role = role;
* }
* public long getSalary() {
* return salary;
* }
* public void setSalary(long salary) {
* this.salary = salary;
* }
* public int getId() {
* return id;
* }
* public void setId(int id) {
* this.id = id;
* }

* }
* package com.journaldev.composition;
* public class Person {
* //composition has-a relationship
* private Job job;
* public Person(){
* this.job=new Job();
* job.setSalary(1000L);
* }
* public long getSalary() {
* return job.getSalary();
* }
* }
* Here is a test class that uses person object and get it’s salary.
* package com.journaldev.composition;
* public class TestPerson {
* public static void main(String[] args) {
* Person person = new Person();
* long salary = person.getSalary();
* }
* }
* **Notice that above test program is not affected by any change in the Job object. If you are looking for code reuse and the relationship between two classes is *has-a* then you should use composition rather than inheritance.**
* **Benefit of using composition is that we can control the visibility of other object to client classes and reuse only what we need.**
* **Also if there is any change in the other class implementation, for example getSalary returning String, we need to change Person class to accomodate it but client classes doesn’t need to change.**
* **Composition allows creation of back-end class when it’s needed, for example we can change PersongetSalary method to initialize the Job object.**

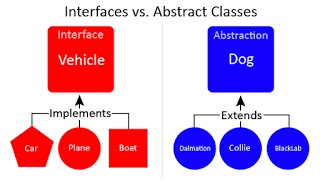
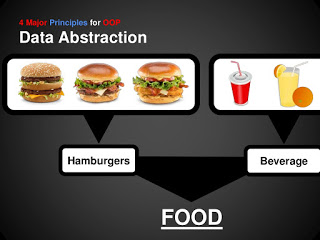
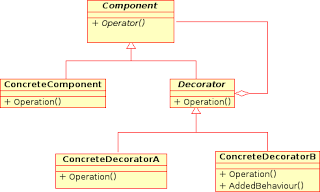
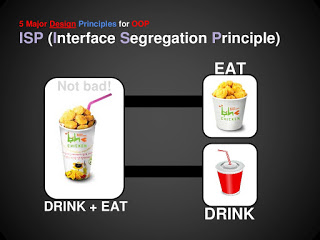
**INTERVIEW QUESTIONS**

* **What is method overloading in OOP or Java?**  
  **When we have multiple methods with the same name but different functionality then it's called method overloading.**
* **For example. System.out.println() is overloaded as we have 6 or 7 println() method each accepting a different type of parameter.**  
    
  **What is method overriding in OOP or Java?**   
  It's one of the magic of object oriented programming where the method is chose based upon an object at runtime.
* In order for method overriding, we need Inheritance and Polymorphism, as we need a method with the same signature in both superclass and subclass.
* **A call to such method is resolved at runtime depending upon the actual object and not the type o variable.**  
  **What is method hiding in Java?**(answer)  
  When you declare two static methods with same name and signature in both superclass and subclass then they hide each other i.e. a call to the method in the subclass will call the static method declared in that class and a call to the same method is superclass is resolved to the static method declared in the super class.  
    
    
  **Is Java a pure object oriented language? if not why? (**[**answer**](http://java67.blogspot.com/2014/03/is-java-pure-object-oriented-programming-language.html)**)  
  Java is not a pure object-oriented programming language e.g. there are many things you can do without objects e.g. static methods.**
* **Also, primitive variables are not objects in Java.**  
  What are rules of method overloading and overriding in Java? ([answer](http://java67.blogspot.sg/2012/09/what-is-rules-of-overloading-and-overriding-in-java.html))  
  One of the most important rule of method overloading in Java is that method signature should be different i.e. either the number of arguments or the type of arguments. Simply changing the return type of two methods will not result in overloading, instead compiler will throw an error. On the other hand, method overriding has more rules e.g. name and return type must be same, method signature should also be same, the overloaded method cannot throw a higher exception etc.   
    
  **The difference between method overloading and overriding?** ([answer](http://java67.blogspot.sg/2012/09/difference-between-overloading-vs-overriding-in-java.html))  
  **Several differences but the most important one is that method overloading is resolved at compile time and method overriding is resolved at runtime.**
* The compiler only used the class information for method overloading, but it needs to know object to resolved overridden method calls. This diagram explains the difference quite well, though:
* [](http://3.bp.blogspot.com/-OCgsVIfnteA/Vmg9Ep7CmII/AAAAAAAAESY/Wk9ERXhb_Tg/s1600/Difference+between+method+overloading+and+overriding+in+Java.gif)
* **Can we overload a static method in Java?** ([answer](http://java67.blogspot.sg/2012/08/can-we-overload-static-method-in-java.html))

Yes, you can overload a static method in Java. You can declare as many static methods of the same name as you wish provided all of them have different method signatures.

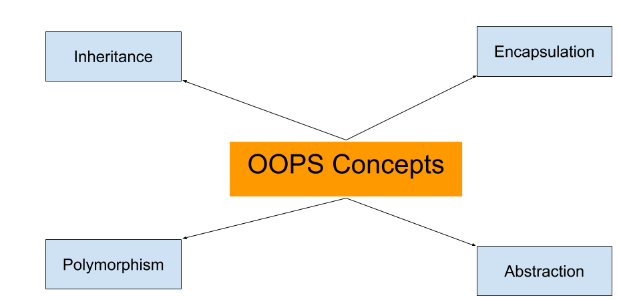
**Can we override static method in Java?**([answer](http://java67.blogspot.sg/2012/08/can-we-override-static-method-in-java.html))  
**No, you cannot override a static method because it's not bounded to an object**.

Instead, static methods belong to a class and resolved at compile time using the type of reference variable. But, Yes, you can declare the same static method in a subclass, that will result in method hiding i.e. if you use reference variable of type subclass then new method will be called, but if you use reference variable of superclass than old method will be called.  
  
  
**Can we prevent overriding a method without using the final modifier?** (answer)  
**Yes, you can prevent the method overriding in Java without using the final modifier. In fact, there are several ways to accomplish it e.g. you can mark the method private or static, those cannot be overridden.**  
  
**Can we override a private method in Java?**([answer](http://java67.blogspot.sg/2013/08/can-we-override-private-method-in-java-inner-class.html))  
**No, you cannot. Since the private method is only accessible and visible inside the class they are declared, it's not possible to override them in subclasses. Though, you can override them inside the inner class as they are accessible there.**  
  
  
**Can we change the return type of method to subclass while overriding?**(answer)  
Yes, you can, but only from Java 5 onward. This feature is known as co-variant method overriding and it was introduced in JDK 5 release. This is immensely helpful if original method return super-class e.g. clone() method return java.lang.Object. By using this, you can directly return the actual type, preventing client-side type casting of the result.  
  
  
**Can we change the argument list of an overriding method? (**[**answer**](http://javarevisited.blogspot.com/2011/08/what-is-polymorphism-in-java-example.html)**)  
No, you cannot. The argument list is part of the method signature and both overriding and overridden method must have the same signature.**  
  
**Can we override a method which throws runtime exception without throws clause?** ([answer](http://javarevisited.blogspot.sg/2011/12/method-overloading-vs-method-overriding.html))  
Yes, there is no restriction on unchecked exception while overriding. On the other hand, in the case of checked exception, an overriding exception cannot throw a checked exception which comes higher in type hierarchy e.g. if original method is throwing IOException than overriding method cannot throw java.lang.Exception or java.lang.Throwable.  
  
  
**How do you call superclass version of an overriding method in sub class? (**answer**)**  
**You can call a superclass version of an overriding method in the subclass by using super keyword. For example to call the toString() method from java.lang.Object class you can call super.toString().**  
  
**Can we override a non-static method as static in Java? (answer)  
Yes, you can override the non-static method in Java, no problem on them but it should not be private or final :)**  
**Can we override the final method in Java?** ([answer](http://javarevisited.blogspot.com/2013/12/when-to-make-method-final-in-java.html))  
**No, you cannot override a final method in Java, final keyword with the method is to prevent method overriding. You use final when you don't want subclass changing the logic of your method by overriding it due to security reason. This is**[**why String class is final in Java**](http://java67.blogspot.com/2014/01/why-string-class-has-made-immutable-or-final-java.html)**. This concept is also used in template design pattern where template method is made final to prevent overriding.**  
**Can we have a non-abstract method inside interface?** (answer)  
From Java 8 onward you can have a non-abstract method inside interface, prior to that it was not allowed as all method was implicitly public abstract. From JDK 8, you can add static and default method inside an interface.  
  
  
**What is an abstract class in Java? (**[answer](http://java67.blogspot.sg/2014/06/why-abstract-class-is-important-in-java.html)**)**  
An abstract class is a class which is incomplete. You cannot create an instance of an abstract class in Java. They are provided to define default behavior and ensured that client of that class should adore to those contract which are defined inside the abstract class. In order to use it, you must extend and implement their abstract methods. BTW, in Java a class can be abstract without specifying any abstract method.  
  
  
**What is an interface in Java? What is the real user of an interface?**([answer](http://java67.blogspot.sg/2014/02/what-is-actual-use-of-interface-in-java.html))  
Like an abstract class, the interface is also there to specify the contract of an API. It supports OOP abstraction concept as it defines only abstract behavior. It will tell that your program will give output but how is left to implementors. The real use of the interface to define types to leverage Polymorphism. See the answer for more detailed explanation and discussion.  
  
  
**The difference between Abstract class and interface?** ([answer](http://java67.blogspot.sg/2012/09/what-is-difference-between-interface-abstract-class-java.html))  
In Java, the key difference is that abstract class can contain non-abstract method but the interface cannot, but from Java 8 onward interface can also contain static and default methods which are non-abstract. See the answer for more detailed discussion as I have described a lot of points there.

* [](http://3.bp.blogspot.com/-iY5H3NVfu14/Vmg99BkXpPI/AAAAAAAAES0/hKRMe87q1P4/s1600/abstract+class+vs+interface+in+Java.png)
* **Can we make a class abstract without an abstract method?** ([answer](http://javarevisited.blogspot.com/2013/04/10-abstract-class-and-interface-interview-question-java-answers.html))  
  Yes, just add abstract keyword on the class definition and your class will become abstract.  
    
   **Can we make a class both final and abstract at the same time? (**[**answer**](http://javarevisited.blogspot.com/2011/12/final-variable-method-class-java.html)**)  
  No, you cannot apply both final and abstract keyword at the class same time because they are exactly opposite of each other. A final class in Java cannot be extended and you cannot use an abstract class without extending and making it a concrete class. As per Java specification, the compiler will throw an error if you try to make a class abstract and final at the same time.**  
    
    
  **Can we overload or override the main method in Java?** ([answer](http://java67.blogspot.com/2015/06/can-you-overload-or-override-main-in-java.html))  
  **No, since main() is a static method, you can only overload it, you cannot override it because the static method is resolved at compile time without needing object information hence we cannot override the main method in Java.**  
  **What is the difference between Polymorphism, Overloading, and Overriding?** ([answer](http://java67.blogspot.sg/2012/10/difference-between-polymorphism-overloading-overriding-java.html))  
  This is slight tricky OOP concept question because Polymorphism is the real concept behind on both Overloading and Overriding. Overloading is compile time Polymorphism and Overriding are runtime Polymorphism.  
    
    
  **Can an interface extend more than one interface in Java?**  
  Yes, an interface can extend more than one interface in Java, it's perfectly valid.  
    
    
  **Can a class extend more than one class in Java?**  
  No, a class can only extend another class because Java doesn't support multiple inheritances but yes, it can implement multiple interfaces.  
    
    
  **What is the difference between abstraction and polymorphism in Java?** ([answer](http://java67.blogspot.sg/2015/05/difference-between-abstraction-and.html))  
  Abstraction generalizes the concept and Polymorphism allow you to use different implementation without changing your code. This diagram explains the abstraction quite well, though:
* [](http://2.bp.blogspot.com/-hBvfoo4bOVY/Vmg9ZhHoQyI/AAAAAAAAESk/FUcbr0i1gDo/s1600/OOP+Abstraction.jpg)
* **What problem is solved by Strategy pattern in Java?** ([answer](http://java67.blogspot.com/2014/12/strategy-pattern-in-java-with-sample.html))
* Strategy pattern allows you to introduce new algorithm or new strategy without changing the code which uses that algorithm. For example, the Collections.sort() method which sorts the list of the object uses Strategy pattern to compare object. Since every object uses different comparison strategy you can compare various object differently without changing sort method.
* **Which OOP concept Decorator design Pattern is based upon?**([answer](http://java67.blogspot.com/2013/07/decorator-design-pattern-in-java-real-life-example-tutorial.html))
* Decorator pattern takes advantage of Composition to provide new features without modifying the original class. A very good to-the-point question for the telephonic round. This is quite clear from UML diagram of Decorator pattern, as you can see the Component is associated with Decorator.
* [](http://2.bp.blogspot.com/-jnzC4Kx48Oc/Vmg-K8LVjqI/AAAAAAAAES8/sYAEghzm688/s1600/Decorator+Design+Pattern+in+Java+UML.png)
* **When to use Singleton design pattern in Java?** ([answer](http://java67.blogspot.com/2012/08/what-is-singleton-pattern-in-java.html))  
  When you need just one instance of a class and wants that to be globally available then you can use [Singleton pattern](http://javarevisited.blogspot.com/2011/03/10-interview-questions-on-singleton.html). It's not free of cost though because it increase coupling between classes and make them hard to test. This is one of the oldest design pattern questions from Java interviews. Please see the answer for more detailed discussion.
* **What is the difference between State and Strategy Pattern?**([answer](http://javarevisited.blogspot.com/2014/04/difference-between-state-and-strategy-design-pattern-java.html))  
  Though the structure or class diagram of State and Strategy pattern is same, their intent is completely different. State pattern is used to do something specific depending upon state while [Strategy](http://java67.blogspot.com/2014/12/strategy-pattern-in-java-with-sample.html) allows you to switch between algorithms without changing the code which uses it.
* **What is the difference between Association, Aggregation, and Composition in OOP?** ([answer](http://javarevisited.blogspot.com/2014/02/ifference-between-association-vs-composition-vs-aggregation.html))  
  When an object is  related to another object it called association. It has two forms, aggregation, and composition. the former is the loose form of association where the related object can survive individual while later is a stronger form of association where a related object cannot survive individually. For example, the city is an aggregation of people but is the composition of body parts.
* **What is the difference between Decorator, Proxy and Adapter pattern in Java?**([answer](http://javarevisited.blogspot.com/2015/01/adapter-vs-decorator-vs-facade-vs-proxy-pattern-java.html))  
  Again they look similar because their structure or class diagram is very similar but their intent is quite different. Decorator adds additional functionality without touching the class, Proxy provides access control and Adapter is used to make two incompatible interfaces work together.
* **What is the 5 objects oriented design principle from SOLID?**([answer](http://javarevisited.blogspot.com/2012/03/10-object-oriented-design-principles.html))  
  SOLID is the term given by Uncle Bob in his classic book, the [Clean Code](http://www.amazon.com/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882?tag=javamysqlanta-20), one of the must-read books for programmers. In SOLID each character stands for one design principle:  
  S for Single Responsibility Principle  
  O for Open closed design principle  
  L for Liskov substitution principle  
  I for Interface segregation principle  
  D for Dependency inversion principle
* [](http://2.bp.blogspot.com/-WwI87RZsjQs/Vmg9bXRGYYI/AAAAAAAAESs/NSOrLHWL670/s1600/OOP+-+Interface+Segregation+Principle.jpg)
* **What is the difference between Composition and Inheritance in OOP?**([answer](http://javarevisited.blogspot.sg/2015/06/difference-between-inheritance-and-Composition-in-Java-OOP.html))
* This is another great OOPS concept question because it test what matters, both of them are very important from a class design perspective. Though, both Composition and Inheritance allows you to reuse code, former is more flexible than later. Composition allows the class to get an additional feature at runtime, but Inheritance is static. You can not change the feature at runtime by substitution new implementation. See the answer for more detailed discussion.
* **COMPOSITION IS MORE FLEXIBE THAN INHERITANCE AS COMPOSITION ALLOWS THE CLASS TO GET AN ADDITIONAL FEATURE AT RUNTIME BUT INHERITANCE IS STATIC. YOU CANNOT CHANGE THE FEATURE AT RUNTIME BY SUBSTITUTION NEW IMPLEMENTATION.**

**Q) What are different oops concept in java?**

Ans) OOPs stands for Object Oriented Programming. The concepts in oops are similar to any other programming languages. Basically, it is program agnostic.



The different OOps concepts are :

* [Polymorphism](http://java-questions.com/oops-interview-questions.html#polymorphism-java)
* [Inheritance](http://java-questions.com/oops-interview-questions.html#inheritance-in-java)
* [Abstraction](http://java-questions.com/oops-interview-questions.html#abstraction)
* [Encapsulation](http://java-questions.com/oops-interview-questions.html#encapsulation)
* [Aggreagation](http://java-questions.com/oops-interview-questions.html#aggregation)
* [Composition](http://java-questions.com/oops-interview-questions.html#composition)
* [Association](http://java-questions.com/oops-interview-questions.html#association)

**Q1) What is polymorphism?**

Ans) The ability to identify a function to run is called Polymorphism. In java, c++ there are two types of polymorphism: compile time polymorphism (overloading) and runtime polymorphism (overriding).

**Mehtod overriding:**Overriding occurs when a class method has the same name and signature as a method in parent class. When you override methods, JVM determines the proper methods to call at the program’s run time, not at the compile time.

**Overloading:**Overloading is determined at the compile time. It occurs when several methods have same names with:

* Different method signature and different number or type of parameters.
* Same method signature but different number of parameters.
* Same method signature and same number of parameters but of different type

Example of Overloading

int add(int a,int b)

float add(float a,int b)

float add(int a ,float b)

void add(float a)

int add(int a)

void add(int a) //error conflict with the method int add(int a)

class BookDetails {

String title;

setBook(String title){}

}

class ScienceBook extends BookDetails {

setBook(String title){} //overriding

setBook(String title, String publisher,float price){} //overloading

}

**Q2) What is inheritance?**

Ans) Inheritance allows a Child class to inherit properties from its parent class. In Java this is achieved by using **extends** keyword. Only properties with access modifier public and protected can be accessed in child class.

public class Parent {

public String parentName;

public String familyName;

protected void printMyName() {

System.out.println(“ My name is “+ chidName+” “ +familyName);

}

}

public class Child extends Parent {

public String childName;

public int childAge;

//inheritance

protected void printMyName() {

System.out.println(“ My child name is “+ chidName+” “ +familyName);

}

}

In above example the child has inherit its family name from the parent class just by inheriting the class. When child object is created printMyName() present in child class is called.

**Q3) What is multiple inheritance and does java support?**

Ans) If a child class inherits the property from multiple classes is known as multiple inheritance. Java does not allow to extend multiple classes. The problem with with multiple inheritance is that if multiple parent classes has a same method name, the at runtime it becomes diffcult for compiler to decide which method to execute from the child class. To overcome this problem it allows to implement multiple Interfaces. The problem is commonly referred as [What is Diamond Problem.](http://java-questions.com/keyConcepts-interview-questions.html#diamond-problem)

**Q) What is difference between polymorphism and inheritance?**

* Inheritance defines parent-child relationship between two classes, polymorphism take advantage of that relationship to add dynamic behaviour in your code.
* Inheritance helps in code reusability by allowing child class to inherit behavior from the parent class. On the other hand Polymorphism allows Child to redefine already defined behaviour inside parent class. Without Polymorphism it's not possible for a Child to execute its own behaviour while represented by a Parent reference variable, but with Polymorphism he can do that.
* Java doesn't allow multiple inheritance of classes, but allows [multiple inheritance of Interface](http://java-questions.com/keyConcepts-interview-questions.html#diamond-problem), which is actually require to implement Polymorphism. For example a Class can be Runnable, Comparator and Serializable at same time, because all three are interfaces. This makes them to pass around in code e.g. you can pass instance of this class to a method which accepts Serializable, or to Collections.sort() which accepts a Comparator.
* Both Polymorphism and Inheritance allow Object oriented programs to evolve. For example, by using Inheritance you can define new user types in an Authentication System and by using Polymorphism you can take advantage of already written authentication code. Since, Inheritance guarantees minimum base class behaviour, a method depending upon super class or super interface can still accept object of base class and can authenticate it.

**Q4) What is an abstraction ?**

Ans) Abstraction is a way of converting real world objects in terms of class. Its a concept of defining an idea in terms of classes or interface. For example creating a class Vehicle and injecting properties into it. E.g

public class Vehicle {

public String colour;

public String model;

}

**Q5) What is Encapsulation?**

Ans) The encapsulation is achieved by combining the methods and attribute into a class. The class acts like a container encapsulating the properties. The users are exposed mainly public methods.The idea behind is to hide how thinigs work and just exposing the requests a user can do.

**Q6) What is Association?**

**Ans) Association is a relationship where all object have their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.**

**Q7) What is Aggregation?**

**Ans) Aggregation is a specialize form of Association where all object have their own lifecycle but there is ownership and child object can not belongs to another parent object. Let's take an example of Department and teacher. A single teacher can not belongs to multiple departments, but if we delete the department teacher object will not destroy. We can think about "has-a" relationship.**

**Q8) What is Composition ?**

Ans) Composition is again specialize form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object does not have their lifecycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room can not belongs to two different house if we delete the house room will automatically delete.

# OOPs concepts – What is Aggregation in java?

BY CHAITANYA SINGH | FILED UNDER: [**OOPS CONCEPT**](http://beginnersbook.com/category/oops-concept/)

Aggregation is a special form of [**association**](http://beginnersbook.com/2013/05/association/). It is also a relationship between two classes like association, however its a **directional** association, which means it is strictly a **one way association.**It represents a **Has-A** relationship.

For example consider two classes Student class and Address class. each student must have an address so the relationship between student and address is a Has-A relationship. But if you consider its vice versa then it would not make sense as an Address doesn’t need to have a Student necessarily. Below example shows this theoretical explanation in a sample java program.  
Student Has-A Address

class Address

{

int streetNum;

String city;

String state;

String country;

Address(int street, String c, String st, String coun)

{

this.streetNum=street;

this.city =c;

this.state = st;

this.country = coun;

}

}

class StudentClass

{

int rollNum;

String studentName;

Address studentAddr;

StudentClass(int roll, String name, Address addr){

this.rollNum=roll;

this.studentName=name;

this.studentAddr = addr;

}

public static void main(String args[]){

Address ad = new Address(55, "Agra", "UP", "India");

StudentClass obj = new StudentClass(123, "Chaitanya", ad);

System.out.println(obj.rollNum);

System.out.println(obj.studentName);

System.out.println(obj.studentAddr.streetNum);

System.out.println(obj.studentAddr.city);

System.out.println(obj.studentAddr.state);

System.out.println(obj.studentAddr.country);

}

}

Output:

123

Chaitanya

55

Agra

UP

India

The above example shows the **Aggregation** between Student and Address classes. You can see that in Student class I have used Address class to obtain student address. Its a typical example of Aggregation in Java.

### Why we need Aggregation?

**To maintain code re-usability**. To understand this lets consider the above example again. Suppose there are two other classes Collegeand Staff along with above two classes Student and Address. In order to maintain Student’s address, College Address and Staff’s address we don’t need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:

Student Has-A Address (Has-a relationship between student and address)

College Has-A Address (Has-a relationship between college and address)

Staff Has-A Address (Has-a relationship between staff and address)

Hence we can improve code re-usability by using Aggregation relationship.

### **Aggregation (HAS-A)**

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

#### Example

class Student

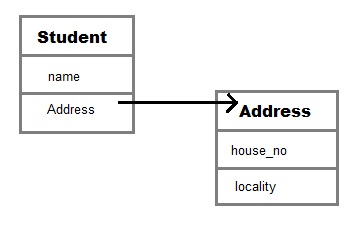
{

String name;

Address ad;

}

Here you can say that **Student** has-a **Address**.



**Student** class has an instance variable of type Address. Student code can use Address reference to invoke methods on the **Address**, and get **Address** behavior.

Aggregation allow you to design classes that follow good Object Oriented practices. It also provide code reusability.

#### Example of Aggregation

class Author

{

String authorName;

int age;

String place;

Author(String name,int age,String place)

{

this.authorName=name;

this.age=age;

this.place=place;

}

public String getAuthorName()

{

return authorName;

}

public int getAge()

{

return age;

}

public String getPlace()

{

return place;

}

}

class Book

{

String name;

int price;

Author auth;

Book(String n,int p,Author at)

{

this.name=n;

this.price=p;

this.auth=at;

}

public void showDetail()

{

System.out.println("Book is"+name);

System.out.println("price "+price);

System.out.println("Author is "+auth.getAuthorName());

}

}

class Test

{

public static void main(String args[])

{

Author ath=new Author("Me",22,"India");

Book b=new Book("Java",550,ath);

b.showDetail();

}

}

**Output :**

Book is Java.

price is 550.

Author is me.

#### Q. What is Composition in java?

Composition is restricted form of Aggregation. For example a class **Car** cannot exist without **Engine**.

class Car

{

private Engine engine;

Car(Engine en)

{

engine = en;

}

}

#### Q. When to use Inheritance and Aggregation?

When you need to use property and behaviour of a class without modifying it inside your class. In such case **Aggregation** is a better option. Whereas when you need to use and modify property and behaviour of a class inside your class, its best to use **Inheritance**.

* In this article we will discuss **Association in Java.** Association establish relationship between two **classes** through their **objects**. The relationship can be one to one, One to many, many to one and many to many.

### Association Example

* class CarClass{
* String carName;
* double carSpeed;
* int carId;
* CarClass(String name, double speed, int Id)
* {
* this.carName=name;
* this.carSpeed=speed;
* this.carId=Id;
* }
* }
* class Driver{
* String driverName;
* int driverAge;
* Driver(String name, int age){
* this.driverName=name;
* this.driverAge=age;
* }
* }
* class TransportCompany{
* public static void main(String args[])
* {
* CarClass obj= new CarClass("Ford", 180.15, 9988);
* Driver obj2 = new Driver("Andy", 45);
* System.out.println(obj2.driverName+" is a driver of car Id: "+obj.carId);
* }
* }
* Output:
* Andy is a driver of car Id: 9988
* In the above example, there is a one to one relationship(**Association**) between two classes: Car and Driver. Both the classes represents two separate entities.

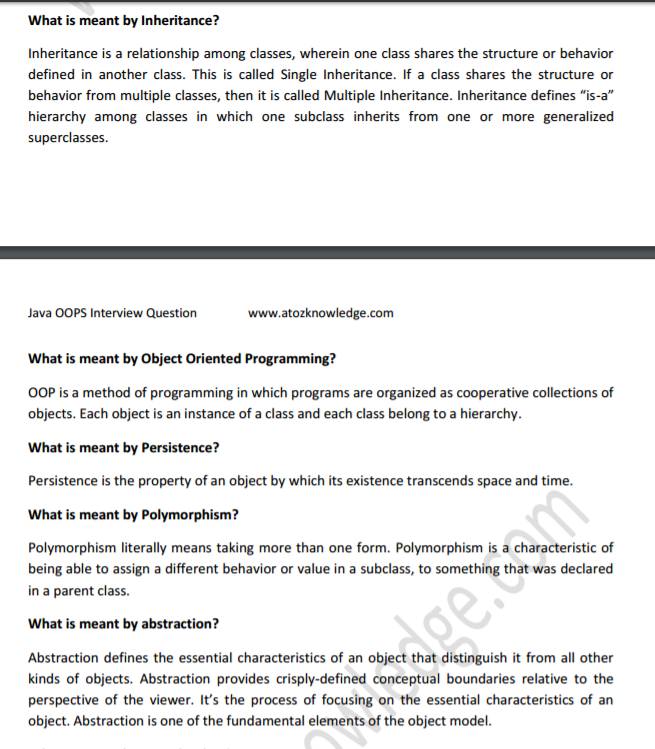
## Association vs Aggregation vs Composition

* Lets discuss **difference between Association, Aggregation and Composition**:
* Although all three are related terms, there are some major differences in the way they relate two classes. **Association** is a relationship between two separate classes which can be of any type say one to one, one to may etc. It joins two entirely separate entities.
* [**Aggregation**](http://beginnersbook.com/2013/05/aggregation/) is a special form of association which is a unidirectional one way relationship between classes (or entities), for e.g. Wallet and Money classes. Wallet has Money but money doesn’t need to have Wallet necessarily so its a one directional relationship. In this relationship both the entries can survive if other one ends. In our example if Wallet class is not present, it does not mean that the Money class cannot exist.
* **Composition** is a restricted form of Aggregation in which two entities (or you can say classes) are highly dependent on each other. For e.g. Human and Heart. A human needs heart to live and a heart needs a Human body to survive. In other words when the classes (entities) are dependent on each other and their life span are same (if one dies then another one too) then its a composition. Heart class has no sense if Human class is not present.
* In this article we will discuss **Association in Java.** Association establish relationship between two **classes** through their **objects**. The relationship can be one to one, One to many, many to one and many to many.

### Association Example

* class CarClass{
* String carName;
* double carSpeed;
* int carId;
* CarClass(String name, double speed, int Id)
* {
* this.carName=name;
* this.carSpeed=speed;
* this.carId=Id;
* }
* }
* class Driver{
* String driverName;
* int driverAge;
* Driver(String name, int age){
* this.driverName=name;
* this.driverAge=age;
* }
* }
* class TransportCompany{
* public static void main(String args[])
* {
* CarClass obj= new CarClass("Ford", 180.15, 9988);
* Driver obj2 = new Driver("Andy", 45);
* System.out.println(obj2.driverName+" is a driver of car Id: "+obj.carId);
* }
* }
* Output:
* Andy is a driver of car Id: 9988
* In the above example, there is a one to one relationship(**Association**) between two classes: Car and Driver. Both the classes represents two separate entities.



  **27. What is an abstraction?**

Abstraction is a good feature of OOPS , and it shows only the necessary details to the client of an object. Means, it shows only necessary details for an object, not the inner details of an object. Example – When you want to switch On television, it not necessary to show all the functions of TV. Whatever is required to switch on TV will be showed by using abstract class.

**28. What are access modifiers?**

Access modifiers determine the scope of the method or variables that can be accessed from other various objects or classes. There are 5 types of access modifiers , and they are as follows:.

* Private.
* Protected.
* Public.
* Friend.
* Protected Friend.