## Delegation

In software engineering, the delegation pattern is a design pattern in object-oriented programming where an object, instead of performing one of its stated tasks, delegates that task to an associated helper object. There is an Inversion of Responsibility in which a helper object, known as a delegate, is given the responsibility to execute a task for the delegator.

## Inheritance v/s delegation

When you want to "copy"/Expose the base class' API, you use inheritance. When you only want to "copy" functionality, use delegation.

When parent class has ‘is-a’ relationship with child class use inheritance.

Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate. Delegation can also a powerful design/reuse technique. The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time. But unlike inheritance, delegation is not directly supported by most popular object-oriented languages, and it doesn’t facilitate dynamic polymorphism.

## Association

Association is a relationship between two objects. In other words, association defines the multiplicity between objects. You may be aware of one-to-one, one-to-many, many-to-one, many-to-many all these words define an association between objects. Aggregation is a special form of association. Composition is a special form of aggregation.

http://javapapers.com/wp-content/uploads/2010/06/association.jpg

**Example:** A Student and a Faculty are having an association.

## Aggregation

Aggregation is a special case of association. A directional association between objects. When an object ‘has-a’ another object, then you have got an aggregation between them. Direction between them specified which object contains the other object. Aggregation is also called a “Has-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/aggregation.jpg

## Composition

Composition is a special case of aggregation. In a more specific manner, a restricted aggregation is called composition. When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition.

http://javapapers.com/wp-content/uploads/2010/06/composition.jpg

**Example:** A class contains students. A student cannot exist without a class. There exists composition between class and students.

### Difference between aggregation and composition

Composition is more restrictive. When there is a composition between two objects, the composed object cannot exist without the other object. This restriction is not there in aggregation. Though one object can contain the other object, there is no condition that the composed object must exist. The existence of the composed object is entirely optional. In both aggregation and composition, direction is must. The direction specifies, which object contains the other object.

***Example:*** A Library contains students and books. Relationship between library and student is aggregation. Relationship between library and book is composition. A student can exist without a library and therefore it is aggregation. A book cannot exist without a library and therefore its a composition. For easy understanding I am picking this example. Don’t go deeper into example and justify relationships!

## Abstraction

Abstraction is specifying the framework and hiding the implementation level information. Concreteness will be built on top of the abstraction. It gives you a blueprint to follow to while implementing the details. Abstraction reduces the complexity by hiding low level details.

***Example:*** A wire frame model of a car.

## Generalization

Generalization uses a “is-a” relationship from a specialization to the generalization class. Common structure and behaviour are used from the specializtion to the generalized class. At a very broader level you can understand this as inheritance. Why I take the term inheritance is, you can relate this term very well. Generalization is also called a “Is-a” relationship.

http://javapapers.com/wp-content/uploads/2010/06/generalization.jpg

***Example:*** Consider there exists a class named Person. A student is a person. A faculty is a person. Therefore here the relationship between student and person, similarly faculty and person is generalization.

## Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

http://javapapers.com/wp-content/uploads/2010/06/realization.jpg

***Example:*** A particular model of a car ‘GTB Fiorano’ that implements the blueprint of a car realizes the abstraction.

## Dependency

Change in structure or behaviour of a class affects the other related class, then there is a dependency between those two classes. It need not be the same vice-versa. When one class contains the other class it this happens.

http://javapapers.com/wp-content/uploads/2010/06/dependency.jpg

***Example:*** Relationship between shape and circle is dependency.

## How does the Object Oriented approach improve software development?

The key benefits are:

1. Re-use of previous work: using implementation inheritance and object composition.
2. Real mapping to the problem domain: Objects map to real world and represent vehicles, customers, products etc: with encapsulation.
3. Modular Architecture: Objects, systems, frameworks etc are the building blocks of larger systems.
4. The increased quality and reduced development time are the by-products of the key benefits discussed above.
5. If 90% of the new application consists of proven existing components then only the remaining 10% of the code have to be tested from scratch.

## Immutable object: Making class immutable

In object-oriented and functional programming, an immutable object is an object whose state cannot be modified after it is created.

Any modification on immutable object will result in another immutable object

Immutable objects are good for caching purpose because you don’t need to worry about the value changes. Other benefit of immutable class is that it is inherently [**thread-safe**](http://www.journaldev.com/1061/java-synchronization-and-thread-safety-tutorial-with-examples), so you don’t need to worry about thread safety in case of multi-threaded environment.

**Benefits of Immutable Classes in Java**

1) Immutable objects are by default [thread safe](http://javarevisited.blogspot.com/2012/01/how-to-write-thread-safe-code-in-java.html), can be shared without synchronization in concurrent environment.

2) Immutable object simplifies development, because its easier to share between multiple threads without external synchronization.

3) Immutable object boost performance of Java application by reducing [synchronization](http://java67.blogspot.com/2013/01/difference-between-synchronized-block-vs-method-java-example.html) in code.  
  
4) Another important benefit of Immutable objects is reusability, you can cache Immutable object and reuse them, much like String literals and Integers.  You can use [static factory methods](http://javarevisited.blogspot.it/2011/12/factory-design-pattern-java-example.html) to provide methods like valueOf(), which can return an existing Immutable object from cache, instead of creating a new one.

Apart from above advantages, immutable object has disadvantage of creating garbage as well

Example: Strings, All wrapper classes like Integer, Double, java.util.Locale etc

Rules:

1) Make class final: Can’t be extended, ensure the class cannot be overridden

2) Make all fields private and final: state can’t be modified

3) No setter methods: state can’t be modified

4) Provide only getter method

5) Public constructor for object creation.

public final class Contacts {

private final String name;

private final String mobile;

public Contacts(String name, String mobile) {

this.name = name;

this.mobile = mobile;

}

public String getName(){

return name;

}

public String getMobile(){

return mobile;

}

}

## Java oops concepts:

1) Inheritance: Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order.

Type: IS-A and Has-A relationships

**When to use**: The subclass Is A super class or Has-A relationship and it makes sense

Don’t use the inheritance for just to reuse the code or they don’t have Is-A relationship.

In case of reuse of code use delegation to make the functionalities to work.

**Why to use**: Avoid duplicate code

Changes to the superclass code instantly reflect in subclass

To promote code reuse if it’s pass is-a test

To use polymorphism

2) Polymorphism: Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.

Type: Overloading and overriding

Explain Overriding with exception during explaining

3) Abstraction: Abstraction refers to the ability to make a class abstract in OOP. An abstract class is one that cannot be instantiated. All other functionality of the class still exists, and its fields, methods, and constructors are all accessed in the same manner. You just cannot create an instance of the abstract class.

* use an abstract class, if you want to provide common implementation to subclasses,
* use an abstract class, if you want to declare non-public members,
* use an abstract class, if you want to be free to add new public methods in the future,

4) Encapsulation: It’s the process of hiding data and restricting access to the data.

Encapsulation means hiding the attributes of a class by marking them private and restricts accessing them through public methods

Benefits *of Encapsulation:*

* The fields of a class can be made read-only or write-only.
* A class can have total control over what is stored in its fields.
* The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

Encapsulation describes the ability of an object to hide its data and methods from the rest of the world and is one of the fundamental principles of object-oriented programming. In Java, a class encapsulates the attributes, which hold the state of an object, and the methods, which define the actions of the object. Encapsulation enables you to write reusable programs. It also enables you to restrict access only to those features of an object that are declared public. All other attributes and methods are private and can be used for internal object processing.

<http://docs.oracle.com/cd/B28359_01/java.111/b31225/chone.htm#BABFJIIC>

### ****Coupling and Cohesion****

Coupling: It means degree in which one class depends on the other.

Cohesion: cohesion is a measure of how the methods of a class or a module are meaningfully and strongly related and how focused they are in providing a well-defined purpose

### ****Explain marker interfaces?****

The marker interface pattern is a design pattern in computer science, used with languages that **provide run-time type information about objects**. It provides **a means to associate metadata with a class where the language does not have explicit support for such metadata.** In java, it is used as interfaces with no method specified.

Example: Serialization, Cloneable, SingleThreadModel.

In AOP ThrowsAdvice

A good example of use of marker interface in java is [Serializable](http://howtodoinjava.com/2012/11/21/a-mini-guide-for-implementing-serializable-interface-in-java/) interface. A class implements this interface to indicate that its non-transient data members can be written to a byte steam or file system.

A major problem with marker interfaces is that an interface defines a contract for implementing classes, and that contract is inherited by all subclasses. This means that you cannot “un-implement” a marker. In the example given, if you create a subclass that you do not want to serialize (perhaps because it depends on transient state), you must resort to explicitly throwing NotSerializableException.

### ****What is the difference between creating String as new() and literal?****

When we create string with new () it’s created in heap and not added into string pool, while String created using literal are created in String pool itself which exists in Perm area of heap.

**Why interface over abstract class?**

* use an abstract class, if you want to provide common implementation to subclasses,
* use an abstract class, if you want to declare non-public members,
* use an abstract class, if you want to be free to add new public methods in the future,
* use an interface if you're sure the API is stable for the long run
* Use an interface if you want to provide the implementing classes the opportunity to inherit from other sources at the same time.
* Most of the time it will be used as reference Type.

The reason for the existence of interfaces in Java: to provide a form of multiple inheritances. In languages with multiple implementation inheritance, an interface would be equivalent to a fully abstract class (a class with only public abstract members).

**Stack and Heap—Quick Review**

* Instance variables and objects live on the heap.
* Local variables live on the stack.

The JVM divided the memory into following sections.

1.Heap

2.Stack

3.Code

4.Static

This division of memory is required for its effective management.

1. The code section contains your bytecode.
2. The Stack section of memory contains methods, local variables and reference variables.
3. The Heap section contains Objects (may also contain reference variables).
4. The Static section contains Static data/methods.

Of all of the above 4 sections, you need to understand the allocation of memory in Stack & Heap the most, since it will affect your programming efforts

**Different ways to create object**

1. Static loading using new operator:

Example:

Car car=new Car();

It will throw NoClassDefFoundException if it fail to find the class at runtime

1. Dynamic loading using class.forName() static method

The above static method returns the class object associated with the class name. The string className can be supplied dynamically at run time.\

String car = "core.Car";

Class carClass = Class.*forName*(car);

Car carObject = (Car) carClass.newInstance();

**How to use Volatile keyword in Java**

Volatile keyword in Java is used as an indicator to Java compiler and  [Thread](http://javarevisited.blogspot.com/2011/02/how-to-implement-thread-in-java.html) that do not cache value of this variable and always read it from [main memory](http://javarevisited.blogspot.sg/2011/05/java-heap-space-memory-size-jvm.html).

**How Hash Map works in Java**

HashMap  works on principle of hashing, we have put() and get() method for storing and retrieving object form HashMap .When we pass an both key and value to put() method to store on HashMap , it uses key object hashcode() method to calculate hashcode and they by applying hashing on that hashcode it identifies bucket location for storing value object. While retrieving it uses key object equals method to find out correct key value pair and return value object associated with that key. HashMap  uses linked list in case of collision and object will be stored in next node of linked list.

Also HashMap  stores both key+value tuple in every node of linked list.

What will happen if two different HashMap  key objects have same hashcode?

They will be stored in same bucket but no next node of linked list. And keys equals () method will be used to identify correct key value pair in HashMap .

In terms of usage Java HashMap is very versatile and I have mostly used HashMap as cache in electronic trading application I have worked . Since finance domain used Java heavily and due to performance reason we need caching HashMap and ConcurrentHashMap  comes as very handy there. You can also check following articles form Javarevisited to learn more about HashMap and Hashtable in Java :

Read more: <http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html#ixzz2fM9jTjwb>

**Singleton class rules**

1. Provide private constructor
2. Static method for providing instance

Example

public class SingletonPattern {

private static SingletonPattern instance = null;

private SingletonPattern() {

// Private constructor to avoid new object creation

}

public static SingletonPattern getInsatnce() {

if (instance == null) {

instance = new SingletonPattern();

}

return instance;

}

}

**Check out 1.5 v/s 1.6 v/s 1.7**

Java 1.5:

1) Enhanced for loop

2) Enumaration

3) Asserssion

4) Autoboxing/Unboxing

5) Generics

6) Annotaions

7) String builder

8) var args

Java 1.6

1. Console
2. Navigable set/map

Java 1.7

1. Exceptions with ||(or)
2. Automatic Null check
3. Resource handling in try block
4. String support for switch

**When an Object becomes Eligible for Garbage Collection**

An Object becomes eligible for Garbage collection or GC if its not reachable from any live threads or any static references in other words you can say that an object becomes eligible for garbage collection if its all references are null. Cyclic dependencies are not counted as reference so if Object A has reference of object B and object B has reference of Object A and they don't have any other live reference then both Objects A and B will be eligible for Garbage collection.   
Generally an object becomes eligible for garbage collection in Java on following cases:  
1) All references of that object explicitly set to null e.g. object = null  
2) Object is created inside a block and reference goes out scope once control exit that block.  
3) Parent object set to null, if an object holds reference of another object and when you set container object's reference null, child or contained object automatically becomes eligible for garbage collection.  
4) If an object has only live references via WeakHashMap it will be eligible for garbage collection. To learn more about HashMap see here [How HashMap works in Java](http://javarevisited.blogspot.com/2011/02/how-hashmap-works-in-java.html).  
Read more: <http://javarevisited.blogspot.com/2011/04/garbage-collection-in-java.html#ixzz2fMhs2dSv>

**What is difference between ArrayList and vector?**

1) Synchronization - ArrayList is not thread-safe whereas Vector is thread-safe. In Vector class each method like add(), get(int i) is surrounded with a synchronized block and thus making Vector class thread-safe.

2) Data growth - Internally, both the ArrayList and Vector hold onto their contents using an Array. When an element is inserted into an ArrayList or a Vector, the object will need to expand its internal array if it runs out of room. A Vector defaults to doubling the size of its array, while the ArrayList increases its array size by 50 percent.

**What is difference between HashMap and HashTable?**

Both collections implements Map. Both collections store value as key-value pairs. The key differences between the two are

1. Hashmap is not synchronized in nature but hashtable is.

2. Another difference is that iterator in the HashMap is fail-safe while the enumerator for the Hashtable isn't.  
**Fail-safe** - if the Hashtable is structurally modified at any time after the iterator is created, in any way except through the iterator's own remove method, the iterator will throw a ConcurrentModificationException.

HashIterator() {

expectedModCount = modCount;// it will be initialized when iterator created and if both mismatched means Hashtable will be structurally modified because this modcount will be used whenever hash map changes like put, remove }

**final** Entry<K,V> nextEntry() {

**if** (modCount != expectedModCount)

**throw** **new** ConcurrentModificationException(); }

3. HashMap permits null values and only one null key, while Hashtable doesn't allow key or value as null.

|  |
| --- |
| **What is difference between List and a Set?** 1) List can contain duplicate values but Set doesn’t allow. Set allows only to unique elements.  2) List allows retrieval of data to be in same order in the way it is inserted but Set doesn’t ensures the sequence in which data can be retrieved.(**Except HashSet**) |
| **What is difference between Arrays and ArrayList ?**  Arrays are created of fix size whereas ArrayList is of not fix size. It means that once array is declared as :   * 1. int [] intArray= new int[6];   2. intArray[7]   // will give ArraysOutOfBoundException.   Also the size of array cannot be incremented or decremented. But with arrayList the size is variable.   1. Once the array is created elements cannot be added or deleted from it. But with ArrayList the elements can be added and deleted at runtime.   List list = new ArrayList(); list.add(1); list.add(3); list.remove(0) // will remove the element from the 1st location.   1. ArrayList is one dimensional but array can be multidimensional.               int[][][] intArray= new int[3][2][1];   // 3 dimensional array   1. To create an array the size should be known or initialized to some value. If not initialized carefully there could me memory wastage. But arrayList is all about dynamic creation and there is no wastage of memory. |

**What is ConcurrentHashMap?**

A concurrentHashMap is thread-safe implementation of Map interface. In this class put and remove method are synchronized but not get method. This class is different from Hashtable in terms of locking; it means that hashtable use object level lock but this class uses bucket level lock thus having better performance.

**What is identityHashMap?**

The IdentityHashMap uses == for equality checking instead of equals(). This can be used for both performance reasons, if you know that two different elements will never be equals and for preventing spoofing, where an object tries to imitate another.

**What is WeakHashMap?**

A hashtable-based Map implementation with weak keys. An entry in a WeakHashMap will automatically be removed when its key is no longer in ordinary use. More precisely, the presence of a mapping for a given key will not prevent the key from being discarded by the garbage collector, that is, made finalizable, finalized, and then reclaimed. When a key has been discarded its entry is effectively removed from the map, so this class behaves somewhat differently than other Map implementations.

**Explain types of references in Java? java.lang.ref package can be used to declare soft, weak and phantom references.**

Garbage Collector won’t remove a **strong reference**.

A **soft reference** will only get removed if memory is low. So it is useful for implementing caches while avoiding memory leaks.

A **weak reference** will get removed on the next garbage collection cycle. Can be used for implementing canonical maps. The java.util.WeakHashMap implements a HashMap with keys held by weak references.

A **phantom reference** will be finalized but the memory will not be reclaimed. Can be useful when you want to be notified that an object is about to be collected.

**What’s difference between comparator and comparable interface in java.**

**Comparator**

1) Its there in java.util.Comparator package |

2) Uses int compare (Object ob1, Object ob2)

3) We build a class separate from the class whose instances we want to sort.

4) We can create many sort sequences.

Example:

import java.util.\*;

class EmployeeSort implements Comparator<Employee> {

public int compare(Employee one, Employee two) {

return one.getName.compareTo(two.getName);

}

}

Collection< Employee> list=new ArrayList< Employee>()

EmployeeSort sortPattern=new EmployeeSort ();

Collection.sort(list, sortPattern);

**Comparable**

1) Its there in java.long.\* package

2) It overrides int compareTo(Object ob) method.

3) We must modify the class whose instances you want to sort.

4) Only one sort sequence can be created.

class Employee implements Comparable {

public int compareTo(Object o) { **// takes an Object rather**

// than a specific type

Employee d = (Employee)o;

return name.compareTo(d.getName());

}}

Comparator is useful if many sort sequences are needed to create and the class whose instance we want to create is final and it can’t be modified.

**Different way of map traversing/iterating:**

Map<String, String> map = **new** HashMap<String, String>();

map.put("babu", "patil");

map.put("shilpa", "biradar");

map.put("krish", "kulk");

**// One type using key set**

Set<String> set = map.keySet();

**for** (String s : set) {

System.*out*.println("key:" + s + " value:" + map.get(s));

}

**// using entryset Map.Entry**

Set<Map.Entry<String, String>> entries = map.entrySet();

**for** (Entry<String, String> ent : entries) {

System.*out*.println("key:" + ent.getKey() + " value:"

+ ent.getValue());

}

**// Noraml**

**for** (String s : map.keySet()) {

System.*out*.println("key:" + s + " value:" + map.get(s));

}

// Iterator

Iterator<String> it = map.keySet().iterator();

**while** (it.hasNext()) {

String key = it.next();

System.*out*.println("using iterator key:" + key + " value:"

+ map.get(key));

}

**Synchronizing map.**

Map m = Collections.synchronizedMap(new HashMap(...));

**Rehashing calculation**

Initial capacity=10

Load factor=.75

Total size=10\*0.75=7.5

Items in hash map >7.5 rehash will take place and it doubles the size.

Iteration over collection views requires time proportional to the "capacity" of the instance (the number of buckets) plus its size (the number of key-value mappings). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

**Setting/Increase JVM heap size**

It is possible to increase heap size allocated by the Java Virtual Machine (JVM) by using command line options.

Following are few options available to change Heap Size.

|  |
| --- |
| -Xms<size>        set initial Java heap size  -Xmx<size>        set maximum Java heap size  -Xss<size>        set java thread stack size |

For example, you can set minimum heap to 64 MB and maximum heap 256 MB for a Java program HelloWorld.

|  |
| --- |
| java -Xms64m -Xmx256m HelloWorld |

**//Get the jvm heap size.**

long heapSize = Runtime.getRuntime().totalMemory();

**String reverse program example.**

* 1. // Reverse string using temp

String word = "babu";

String temp = **new** String();

**int** i = 0;

**for** (i = word.length() - 1; i >= 0; i--) {

temp += word.charAt(i);

}

System.*out*.println("word " + temp);

* 1. // Reverse string without using temp

String word2 = **new** String("babu");

**int** k = word2.length() - 1;

**for** (**int** j = word2.length() - 1; j >= 0; j--) {

word2 += word2.charAt(j);

}

System.*out*.println(word2.substring(k + 1));

**Checked v/s Unchecked exceptions**

Exception : Checked exceptions need to be declared in a method or constructor's (using throws} clause if they can be thrown by the execution of the method or constructor and propagate outside the method or constructor boundary.

Checked Exception: Handled by developers and thrown during compile time.

Unchecked Exception: Runtime time exceptions.

Exceptions with polymorphism

Checked exceptions always should be subclass exceptions of the base class methods defined exception.

Unchecked exception doesn’t have any constraints with polymorphic behavior

Checked exceptions with constructor is reverse of polymorphic methods

Use checked exceptions when the client code can take some useful recovery action based on information in exception. Use unchecked exception when client code cannot do anything. For example, convert your [SQLException](http://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html) into another checked exception if the client code can recover from it and convert your [SQLException](http://docs.oracle.com/javase/7/docs/api/java/sql/SQLException.html) into an unchecked (i.e. [RuntimeException](http://docs.oracle.com/javase/7/docs/api/java/lang/RuntimeException.html)) exception, if the client code cannot do anything about it.

use checked exceptions for recoverable conditions and unchecked exceptions for programming errors

**Finally block test:**

Finally block will not execute only in case of system.exit(0);

**try**{

System.*exit*(0);

}

**catch**(Exception e){

System.*out*.println("in catch");

}

**finally**{

System.*out*.println("in finally");

}

Finally block will execute in case of **return**

**try**{

**return**;

}

**catch**(Exception e){

System.*out*.println("in catch");

}

**finally**{

System.*out*.println("in finally");

}

**Using class.forName()**

The main reason for loading jdbc driver using Class.forName() is, the driver can change dynamically. In the static block all Drivers will create an instance of itself and register that class with DriverManager using DriverManager.registerDriver() method.

Since the Class.forName(String className) by default resolve the class, it will initialize the static initializer.

So when we call Class.forName("com.sun.jdbc.odbc.JdbcOdbcDriver"),

the Driver class will be loaded, instantiated and registers with DriverManager

So if we are using new Operator we have to do the following things.

Driver drv = new com.sun.jdbc.odbc.JdbcOdbcDriver();

DriverManager.registerDriver(drv);

**Thread deadlock example.**

**public** **class** DeadlockRisk {

**private** **static** **class** Resource {

**public** **int** value;

}

**private** Resource resourceA = **new** Resource();

**private** Resource resourceB = **new** Resource();

**public** **int** read() {

**synchronized** (resourceA) { // May deadlock here

**synchronized** (resourceB) {

**return** resourceB.value + resourceA.value;

}

}

}

**public** **void** write(**int** a, **int** b) {

**synchronized** (resourceB) { // May deadlock here

**synchronized** (resourceA) {

resourceA.value = a;

resourceB.value = b;

}

}

}

}

**What are design patterns?**

* Design patterns are not algorithms.
* These are language independent strategies for solving common object-oriented design problems.
* These patterns provide efficient and effective solutions to common problems in software development.
* Design pattern are based on the base principles of object orientated design.
  1. Program to an 'interface', not an 'implementation'.
  2. Favor 'object composition' over 'class inheritance'

*1. Composition is easily achieved at runtime while inheritance provides its features at compile time*  
 *2. Composition is also known as HAS-A relation and inheritance is also known as IS-A  relation*

**Design Patterns can be divided into three parts**

* 1. **Creational Patterns:** This pattern is used to create object in a form that they can follow the architecture principle of loose coupling, encapsulation and abstraction
  2. **Structural Patterns:** This pattern is used to form larger object structures from many different objects
  3. **Behavioural Patterns:** This pattern is used to manage relationships, interaction, algorithms and responsibilities between various objects

**Why use factory pattern or abstract factory pattern?**

Factory pattern returns an instance of several (product hierarchy) subclasses (like Circle, Square etc), but the calling code is unaware of the actual implementation class. The calling code invokes the method on the interface (for example Shape) and using polymorphism the correct draw() method gets invoked. So, as you can see, the factory pattern reduces the coupling or the dependencies between the calling code and called objects like Circle, Square etc. This is a very powerful and common feature in many frameworks. You do not have to create a new Circle or a new Square on each invocation as shown in the sample code, which is for the purpose of illustration and simplicity. In future, to conserve memory you can decide to cache objects or reuse objects in your factory with no changes required to your calling code. You can also load objects in your factory based on attribute(s) read from an external properties file or some other condition. Another benefit going for the factory is that unlike calling constructors directly, factory patterns have more meaningful names like getShape(…), getInstance(…) etc, which may make calling code more clear.

**OPEN CLOSED Principal:**

Software should be open to extension but closed to modification. Behavior of class ,method ,interface should be extensible but should be available for internal modification. Abstraction is the way to apply this . Subclass of abstract class should be free to extend the features of abstract class But should be restricted to modify existing method or behavior.