**SOLID**

**Single Responsibility principal - only one responsiblity and reason to change**

**Open- closed principal - A class should be open for extention and closed for modification**

**LisKove subsitution principal - Subtypes should be replaceble with their base types**

**Interface segration principal - Many specific interfaces are better then a general interfaces**

**Dependency inversion principal - we must depend on abstractions and not concreate classes; High level modules should not be depends on low level modules and both should depend on abstractions**

**https://www.youtube.com/watch?v=GATSXm7WAxU**

**Dependency injection - a class should concentrate on fulfilling it responsiblities, not on creating objects that are required to fulfill them, and these objects are provided by DI**

**reduces the boiler plate and duplcate ccode as the dependencies are provided by the injector.**

**your program is easuer ti test as the dependencies ca be isolated and mocked, allowing components to communicate through contracts.**

**construction injection**

**setter injection**

**Field injection**

**Inversion control : principal in software engineering which transfer the control of objects or portions of program to a framework**

**- enables a framework to take a control over the flow of program and make calls to our custom code**

**- to do that framework use abstractions, hence to add an extra behavior, we extend the classes of the framework**

**Creational pattern: give more flexibility in how the objects are actully created**

**1.Singleton pattern - let you access your object from anywhere in your application by creating only one instance of the object -**

**create a private constructor to block others to create the instance of the class**

**Intalise Same object instance in the class and make it volitile**

**create a getinstance method by checking the instance null and syncornising the logic to create object and if availabe retun the same object**

**2. Factory Method pattern - loosens the coupling of a given code by separating the product's construction code from the code that uses this product**

**- relies heavily on inheritence**

**- lets creator-subclasses decide whihc class to instantiate**

**3.Abstract Factory pattern - extention of factory method**

**4.Builder pattern - Constructor - optimastion**

**5.The Prototype Pattern - delegates the object duplication or cloning process to the actual objects that are being cloned**

**usage - your code should not depend on the concreate classess of the objects that you need to copy or duplicate**

**will clone objects without coupling then to their concreate classes**

**will get rid of repeted initialization code**

**Structural pattern: deals with how inheritence and compostion can be used to provide extra functionality**

**6.Adapter Pattern : uses inheritence and composition to enable objects with incompitable interfaces to collaborate with one another**

**It create a middle-layer class that serves as a translator**

**the adopting behavior is now saperated and we can introduce new adopters without breaking existing code**

**these two hierarchies are refered as abstractions and implementation**

**7.Bridge Pattern - spilts a large class into two separate hierarchies which can be developed independently**

**8.Composit Pattern - object graph**

**9.Decorator Pattern - attach new behaviors to an object by placing this object inside spacial wrapper object that contain the behaviors**

**10.Facade Pattern - a class server as a front facing interface and mask complex underlying structural code**

**imporves readabilty and usablity of software lib by hiding the interaction of its components**

**define entry points to each level of subsystem thus decoupling muliple subsystem and communicating them only through facede**

**11.Flyweight pattern : refers to an object that minimise memory usage by sharing some of the initial objects data with similar object**

**It should be used when your huge program must support a huge number of similar objects whihc barely fits into the available amount of RAM**

**spilting the state ot the initial object into two. an intersic-immutable state and extrinsic-mutable one**

**12.Proxy pattern : allows you to control the access of particular object by performing somthing before or after the request reaches that object. the proxy pattern implement the same interface of the original object. Manages the life cycle of the service object and the proxy will work even if the service object is not ready or is not available**

**Behavioral Design Patterns - about communication and assignment of responsiblities between our objects**

**13.The Chain of Responsibility Pattern - transforms particular behavior into stand-alone objects called handler**

**request - handler - handler - handler - solved**

**14.Command Pattern - turns a request or a behaviour into a stand-alone object that contains everything about the request**

**encapsulates all the relevent information needed to perform an action or trigger an event**

**The Template Method Pattern**

**- truns an algorithm into a series of individual methods by intacting the structure of the base algorithm**

**- eliminates code duplication by pulling up the steps with similar implemtnation into the superclass houing the template method**

**- the code that varies in split between different implemtnations**

**15.The Mediator Pattern - Airport traffic control room - mediator desing pattern**

**- defines an object that encapsulates how a set of other objects interact with each other**

**- restircts direct communications between objects and forces them to colloborate via a mediator, hence reducing the dependencies between them**

**16.The Memento Pattern -**

**let you save and restore the state of an object without revealing the details of its implementation**

**to do that this pattern makes full copies of an object's state**

**makes the original objects data safe and secure**

**delegates the creating of the objects state snapshot to the object itself**

**17.The Observer Pattern - subscription mechanism and notify ---**

**Allows you to change or take action on a set of objects when and if the state another object changes**

**Notification service -> has infrastructure to maintain**

**Subscribers -> Event listener**

**18.The State Pattern - state machine - state can be dependent as you can easily jump from one state to another**

**19.Strategy design pattern - define a family of algorithms, puts each of then in a separate class, and makes their objects interchangeable.**

**20. The Iterator Pattern**

**21.The Visitor Pattern - isolates particular behaviors from the objects on which they operate, and places them in a single class**

**classes will be more focused on their main job hence enforcing the SRP**

**What is the difference between aggregation and composition?**

**Aggregation Composition**

Aggregation is an association in which one class belongs to a collection. This is a part of a whole relationship where a part can exist without a whole.

**For example** a line item is a whole and product is a part. If a line item is deleted then corresponding product need not be deleted. So **aggregation has a weaker relationship**.

Composition is an association in which one class belongs to a collection. This is a part of a whole relationship where a part cannot exist without a whole. If a whole is deleted then all parts are deleted. **For example** An order is a whole and line items are parts.

If an order deleted then all corresponding line items for that order should be deleted. So **composition has a stronger relationship**.

What is the difference between an abstract class and an interface and when should you use them? **LF DP DC**

**A 10:** In design, you want the base class to present *only* an interface for its derived classes. This means, you don’t want anyone to actually instantiate an object of the base class. You only **want to upcast to it** (implicit upcasting, which gives you polymorphic behaviour), so that its interface can be used. This is accomplished by making that class

*abstract*using the **abstract** keyword. If anyone tries to make an object of an **abstract** class, the compiler prevents

it.

The **interface** keyword takes this concept of an **abstract** class a step further by preventing any method or function

implementation at all. You can only declare a method or function but not provide the implementation. The class,

which is implementing the interface, should provide the actual implementation. The **interface** is a very useful and

commonly used aspect in OO design, as it provides the **separation of interface and implementation** and

enables you to:

􀂃Capture similarities among unrelated classes without artificially forcing a class relationship.

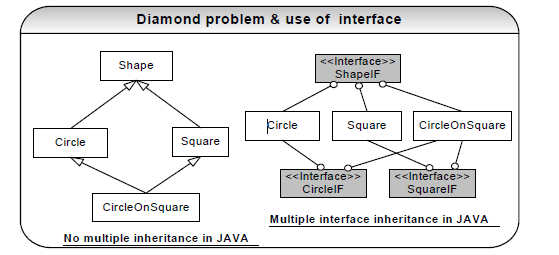
􀂃Declare methods that one or more classes are expected to implement.

􀂃Reveal an object's programming interface without revealing its actual implementation.

􀂃Model multiple interface inheritance in Java, which provides some of the benefits of full on multiple

inheritances, a feature that some object-oriented languages support that allow a class to have more than one

superclass.



**Abstract class Interface**

Have executable methods and abstract methods. Have no implementation code. All methods are abstract.

Can only subclass one abstract class.

A class can implement any number of interfaces.

Can have instance variables, constructors and any

visibility: public, private, protected, none (aka package).

Cannot have instance variables, constructors and can have

only public and none (aka package) visibility.

**When to use an abstract class?**: In case where you want to use **implementation inheritance** then it is usually

provided by an abstract base class. Abstract classes are excellent candidates inside of application frameworks.

Abstract classes let you define some default behaviour and force subclasses to provide any specific behaviour.

Care should be taken not to overuse implementation inheritance as discussed in **Q8** in Java section.

**When to use an interface?:**For polymorphic interface inheritance, where the client wants to only deal with a type

and does not care about the actual implementation use interfaces. If you need to change your design frequently,

you should prefer using interface to abstract. **CO** Coding to an interface **reduces coupling** and interface

inheritance can achieve **code reuse** with the help of **object composition**. Another justification for using interfaces

is that they solve the ‘**diamond problem**’ of traditional multiple inheritance as shown in the figure. Java does not

support multiple inheritances. Java only supports **multiple interface inheritance**. Interface will solve all the

ambiguities caused by this ‘diamond problem’.

**Design pattern:** Strategy design pattern lets you swap new algorithms and processes into your program without

altering the objects that use them. **Strategy design pattern**: Refer **Q11** in How would you go about… section.

Why there are some interfaces with no defined methods (i.e. marker interfaces) in Java? **LF**

**A 11:** The interfaces with no defined methods act like markers. They just tell the compiler that the objects of the classes implementing the interfaces with no defined methods need to be treated differently. ***Example*** Serializable

Cloneable etc

How does Java allocate stack and heap memory? Explain re-entrant, recursive and idempotent

methods/functions?

Each time an object is created in Java it goes into the area of memory known as **heap**. The primitive variables like

int and double are allocated in the **stack**, if they are local method variables and in the **heap** if they are member

variables (i.e. fields of a class). In Java methods local variables are pushed into stack when a method is invoked

and stack pointer is decremented when a method call is completed. In a multi-threaded application each thread

will have its own stack but will share the same heap. This is why care should be taken in your code to avoid any

concurrent access issues in the heap space. The stack is threadsafe (each thread will have its own stack) but the

heap is not threadsafe unless guarded with synchronisation through your code.

A method in stack is **re-entrant** allowing multiple concurrent invocations that do not interfere with each other. A

function is **recursive** if it calls itself. Given enough stack space, recursive method calls are perfectly valid in Java

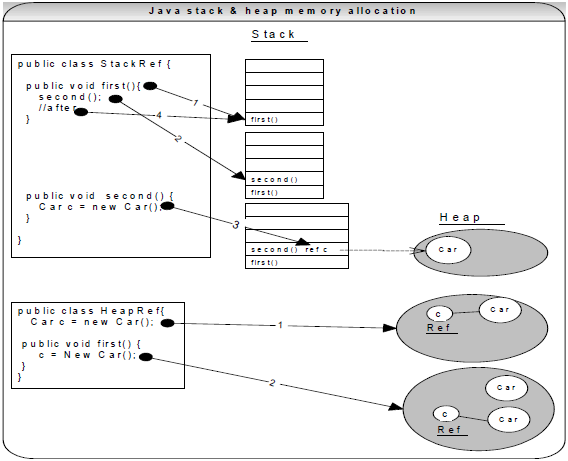
though it is tough to debug. Recursive functions are useful in removing iterations from many sorts of algorithms. All

recursive functions are re-entrant but not all re-entrant functions are recursive. **Idempotent** methods are methods,

which are written in such a way that repeated calls to the same method with the same arguments yield same

results. For example clustered EJBs, which are written with idempotent methods, can automatically recover from a

server failure as long as it can reach another server.



What do you know about the Java garbage collector? When does the garbage collection occur? Explain different

types of references in Java?

Each time an object is created in Java, it goes into the area of memory known as heap. The Java heap is called

the garbage collectable heap. The garbage collection **cannot be forced**. The garbage collector runs in low

memory situations. When it runs, it releases the memory allocated by an unreachable object. The garbage

collector runs on a low priority daemon (background) thread. You can **nicely ask** the garbage collector to collect

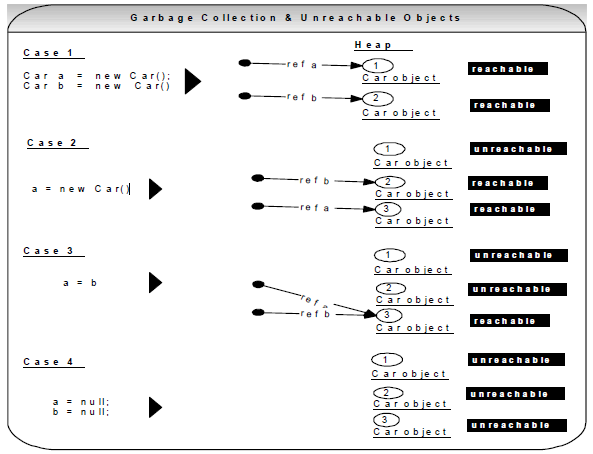
garbage by calling *System.gc()* but **you can’t force it**.

***What is an unreachable object?*** An object’s life has no meaning unless something has reference to it. If you

can’t reach it then you can’t ask it to do anything. Then the object becomes unreachable and the garbage collector

will figure it out. Java automatically collects all the unreachable objects periodically and releases the memory

consumed by those unreachable objects to be used by the future reachable objects.



We can use the following options with the **Java** command to enable tracing for garbage collection events.

**-verbose:gc**reports on each garbage collection event.

***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.

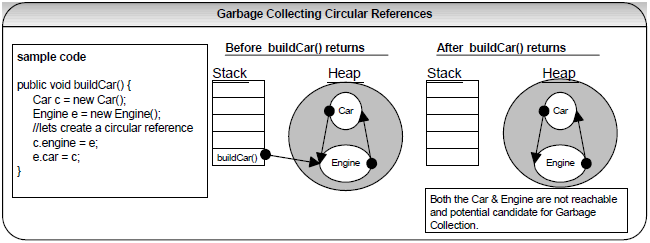
􀂃A***phantom reference*** will be finalized but the memory will not be reclaimed. Can be useful when you want to

be notified that an object is about to be collected.

If you have a circular reference of objects, but you no longer reference it from an execution thread, will this object

be a potential candidate for garbage collection?

Yes. Refer diagram below.



What is a **singleton** pattern? How do you code it in Java?

A singleton is a class that can be instantiated **only one time in a JVM per class loader**. Repeated calls always

return the same instance. Ensures that a class has only one instance, and provide a **global point of access**. It

can be an issue if singleton class gets loaded by multiple class loaders.

public class OnlyOne {

**private static** OnlyOne one = new OnlyOne();

**private**OnlyOne(){… } **//private constructor.** This class cannot be instantiated from outside.

**public**static OnlyOnegetInstance() {

return one;

}

}

**To use it:**

//No matter how many times you call, you get the same instance of the object.

OnlyOnemyOne = **OnlyOne.getInstance();**

**Note:** The constructor must be explicitly declared and should have the private access modifier, so that it cannot be

instantiated from out side the class. The only way to instantiate an instance of class *OnlyOne*is through the

**getInstance()** method with a public access modifier.

**When to use:** Use it when only a single instance of an object is required in memory for a single point of access.

For example the following situations require a **single point of access**, which gets invoked from various parts of

the code.

􀂃Accessing application specific properties through a singleton object, which reads them for the first time from

a properties file and subsequent accesses are returned from in-memory objects. Also there could be

another piece of code, which periodically synchronizes the in-memory properties when the values get

modified in the underlying properties file. This piece of code accesses the in-memory objects through the

singleton object (i.e. global point of access).

􀂃Accessing in-memory object cache or object pool, or non-memory based resource pools like sockets,

connectionsetc through a singleton object (i.e. global point of access).

**What is the difference between a singleton class and a static class?** Static class is one approach to make a class singleton

by declaring the class as “final” so that it cannot be extended and declaring all the methods as static so that you can’t create any

instance of the class and can call the static methods directly.

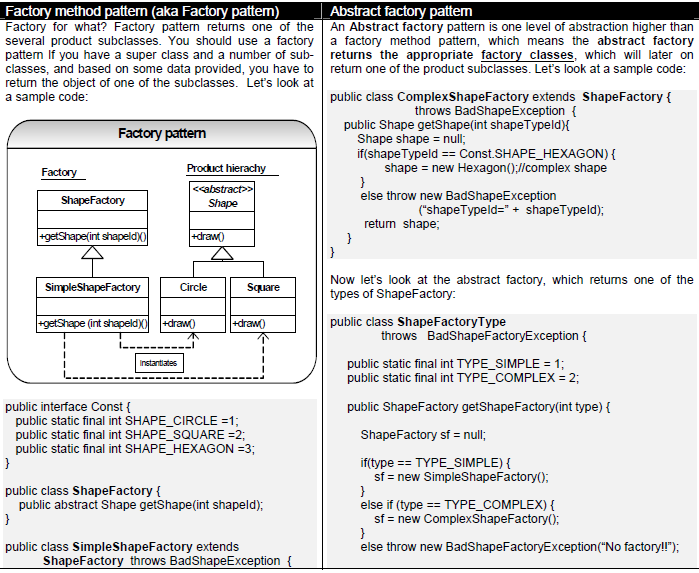
What is a factory pattern?

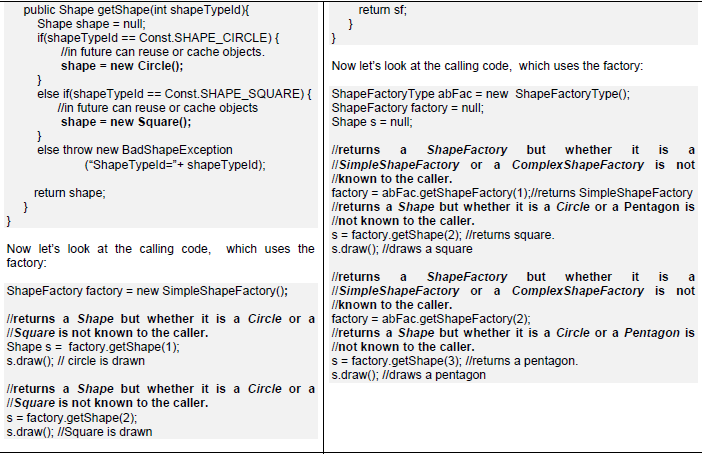
A **Factory method pattern** (aka **Factory pattern**) is a creational pattern. The creational patterns abstract the

object instantiation process by hiding how the objects are created and make the system independent of the object

creation process. An **Abstract factory** pattern is one level of abstraction higher than a factory method pattern,

which means it returns the factory classes.





**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 [Refer **Q8** in Java section for polymorphism]. 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.

**Can we use the singleton pattern within our factory pattern code?** Yes. Another important aspect to consider

when writing your factory class is that, it does not make sense to create a new factory object for each invocation

as it is shown in the sample code, which is just fine for the illustration purpose.

ShapeFactory factory = new SimpleShapeFactory()**;**

To overcome this, you can incorporate the singleton design pattern into your factory pattern code. The singleton

design pattern will create only a single instance of your *SimpleShapeFactory*class. Since an abstract factory

pattern is unlike factory pattern, where you need to have an instance for each of the two factories (i.e.

*SimpleShapeFactory*and ComplexShapeFactory) returned, you can still incorporate the singleton pattern as an

access point and have an instance of a *HashMap,* store your instances of both factories. Now your calling method

uses a static method to get the same instance of your factory, hence conserving memory and promoting object

reuse:

ShapeFactory factory = ShapeFactory. Ge/tFactoryInstance();

factory.getShape();

8.

9.

Creation patterns

**Factory method/abstract factory design pattern**: popular pattern, which gets frequently asked in interviews.

**Singleton pattern**: popular pattern, which gets frequently asked in interviews.

Behavioral Strategy patterns

**Visitor design pattern**: to avoid instanceof and typecast constructs.

**Strategy design pattern**: A popular design pattern used by AWT layout managers.

**Command pattern**: used by Swing action architecture and also a popular design pattern.

**Decorator design pattern**: used by Java I/O API. A popular design pattern.

**Reactor design pattern/Observer design pattern**: used by Java NIO API.

Structural pattern

**Composite design pattern**: used by GUI components and also a popular design pattern

**MVC design pattern/architecture**: used by Swing components and also a popular pattern.