## 1. Explain polymorphism and why it is important.

Polymorphism is one of the core principles of OOP and describes the situation in which different objects behave similarly. It describes the concept that allows to access objects of different types through the same interface where each of the types can provide their own independent implementations. A subclass can override a method of its superclass, and if a subclass is overridden JVM calls the overridden implementation. It allows to reuse the code and build robust software.

## 2. Explain the open close principle and give an example.

The principle is quote as “Components of software should be open for extension, but closed for modifications”

If our software component is class, then Classes should be open for extension, but closed for modification. According to the principle, we should not modify the existing code that was working before. “Open to extension” in this context mean that, classes should be designed in such a way that new functionalities can be added as new requirement are generated.We can take a simple example of different polygon in this case.

For example, consider the following piece of code,

*void* calculateArea(Rectangle rectangle) {  
 *return* rectangle.length \* rectangle.breadth;  
}

The code calculates the area of rectangle. Now, if we need to add a new method to calculate an area of a different shape, Circle we would need to add a separate implementation for the method, which might look like this,

*void* calculateArea(Circle circle) {  
 *return* Math.PI \* Math.pow(circle.radius, 2);  
}

Now, if we want to add more shapes to our code, we need to modify the existing code and add more methods like such. This we can create an interface and each of the Shape classes Circle / Rectangle and Pentagon can implement the interface which makes it much easier to add any new code without modifying the existing code.

## 3. Explain early binding and when it is possible.

In case of early binding compiler maps the method at compile-time. It is also known as static binding. The compiler determines the type of object and resolves them already during the compile time of program. The binding of static, private and final methods are always done during the compile-time with early (static) binding since they cannot be overridden. Early binding is much faster since the information needed to call a method is available before run time.

## 4. Explain late binding and why it is needed.

In case of dynamic binding or late binding, compiler resolves the method calls during the execution of the program. The type of object is determined during the execution of the program, and thus it is called dynamic binding. The major advantage of late binding is its flexibility since a single method can handle many different types of objects during the runtime. This helps in reusing the codebase. Dynamic binding is achieved with major OOP principles such as inheritance and interface.

## 5. Explain programming to an interface and what are the advantages of doing so.

The beauty of interfaces is that a piece of code might get an object and expect a certain behavior from it, but it does not care about what kind of object it is, only that it supports the behavior needed. List interface in java can be taken as an example. All we need to care is the structure that can contain multiple data items that can be accessed through iteration. If we know we need constant insert/delete from end of the list, we might pick LinkedList. If we know we need random access by index, we might pick up ArrayList concrete class.

## 6. Explain Factory design pattern and why is it important

The Factory method pattern suggest replacing the direct object construction calls (using public constructor) with calls to a special factory method. It is easier to extend product construct code independently from the rest of the code. For example, if a new product type is added. When dealing with large, resource-intensive objects, we need to keep track of all created object and connect them together before they can be used. Often, there can be duplication in code so, instead of using the constructor to do setup code, we can delegate it to factory method which makes the code clean and helps in maintainability, reusability.

## 7. List at three advantages of using a Factory method over using the constructor

* Helps in avoiding tight coupling between the creator and the concreate products.
* Single Responsibility Principle: The product creation code can be moved into one place in the program, making the code easier to support.
* Open/Closed Principle: The new types of products can be introduced into the program without breaking existing client code.

## 8. Explain Template Method design pattern and how it is useful

Template method is a behavioral design pattern that defines the skeletal of an algorithm in the superclass but let’s subclass override specific steps of algorithm without changing its structure.

Template method pattern can be used when you want client code to extend only particular steps of an algorithms, but not the whole algorithms or structure. The pattern also removes the redundant code when there are several classes with identical algorithms with some minor differences.

## 9. Explain Listener design pattern and give an example of its application

Observer / listener design pattern is a behavior design pattern. It is used in event driven subsystem to notify some object about certain events. For example, in a UI driven system, a button may send click event to a subscriber of the event. The pattern usually has two different parties, a publisher and subscriber object. Subscriber subscribes to published for changes and publisher notifies subscriber when the change happens.

## 10. Explain the Façade design pattern and give an example of how it is useful for information hiding in subsystem design

Façade is a class that provides a simple interface to a complex subsystem which contains a lot of moving parts. A façade might provide limited functionality in comparison to working with the subsystem directly. However, it includes only those features that clients really care about. Having a façade, helps to easily integrate a sophisticated library with simple interfaces. Suppose and app needs upload short funny videos with cats to social media. It could potentially use a professional video conversion library. However, all that is really needs is a class with the single method encode(filename, format).

## 11. Explain The Singelton design pattern and show how you implement it.

The singleton pattern is a design pattern that restricts the instantiation of a class to one object. For example, a single DB connection is shared by multiple objects as creating a separate DB connection for every object may be costly. Similarly, there can be a single configuration manager or error manager in an application that handles all problems instead of creating multiple managers.

Here is a simple implementation of singleton.

*class* Singleton  
{  
 *private static* Singleton obj = *new* Singleton();  
  
 *private* Singleton() {}  
  
 *public static* Singleton getInstance()  
 {  
 *return* obj;  
 }  
}