Creational Patterns, as the name implies, are most concerned about solutions and options revolving around instantiating objects, and how to do so more efficiently in the most varied of circumstances.

Def 2: Creational patterns often used in place of direct instantiation with constructors. They make the creation process more adaptable and dynamic. In particular, they can provide a great deal of flexibility about which objects are created, how those objects are created, and how they are initialized.

The **Singleton pattern** enables an application to have one and only one instance of a class per JVM.

The **Builder pattern** is used to help build final objects, for classes with a huge amount of fields or parameters in a step-by-step manner. It's not very useful in small, simple classes that don't have many fields, but complex objects are both hard to read and maintain by themselves.Initializing an object with more than a few fields using a constructor is messy and susceptible to human error.

Def 2: Builder design pattern is an alternative way to construct complex objects and should be used only when we want to build different types of immutable objects using the same object building process.

The **Prototype pattern** is used in scenarios where an application needs to create a large number of instances of a class, which have almost the same state or differ very little, then the Prototype pattern is used mainly to minimize the cost of object creation, usually when large-scale applications create, update or retrieve objects which cost a lot of resources.

This is done by copying the object, once it's created, and reusing the copy of the object in later requests,to avoid performing another resource-heavy operation. It depends on the decision of the developer whether this will be a full or shallow copy of the object, though the goal is the same.

The **Factory Method**, also often called the **Factory Pattern** is a widely used design pattern that commands object creation. In this pattern, a Factory class is created as the parent class of all sub-classes belonging to a certain logical segment of related classes.

Just like a 'SessionFactory' is used to create, update, delete and manipulate all 'Session' objects, so is any other factory responsible for their set of child classes.

It's important to note that the sub-classes can't be reached without using their respective factory. This way, their creation is both hidden from the client and is dependent on the factory.

The **Abstract Factory pattern** builds upon the Factory Pattern and acts as the highest factory in the hierarchy. It represents the practice of creating a 'factory of factories'.

This pattern is responsible for creating all other factories as its sub-classes, exactly like how factories are responsible for creating all of their own sub-classes.

Structural Patterns are concerned about providing solutions and efficient standards regarding class compositions and object structures. Also, they rely on the concept of inheritance and interfaces to allow multiple objects or classes to work together and form a single working whole.

The **Adapter pattern**, as the name implies, adapts one interface to another. It acts as a bridge between two unrelated, and sometimes even completely incompatible interfaces, similar to how a scanner acts as a bridge between a paper and a computer.

Def 2: An adapter converts the interface of a class into another interface clients expect. It lets classes work together that couldn't otherwise because of incompatible interfaces.

The **Bridge pattern** is used to segregate abstract classes from their implementations and act as a bridge between them. This way, both the ***abstract*** class and the ***implementation*** can change structurally without affecting the other.

Def 2: Bridge design pattern is used to decouple a class into two parts – *abstraction* and it's *implementation* – so that both can evolve in future without affecting each other. It increases the loose coupling between class *abstraction* and its *implementation*.

The **Composite pattern** is used when we need a way to treat a whole group of objects in a similar, or the same manner. This is usually done by the class that "owns" the group of objects and provides a set of methods to treat them equally as if they were a single object.

Def 2: Composite design pattern helps to compose the objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

The **Decorator pattern** is used to alter an individual instance of a class at runtime, by creating a decorator class which wraps the original class this way, changing or adding functionalities of the decorator object won't affect the structure or the functionalities of the original object.

It differs from classic inheritance in the fact that it's done at runtime, and applies only to an individual instance, whereas inheritance will affect all instances, and is done at compile time.

Def 2: Decorator design pattern is used to add additional features or behaviors to a particular instance of a class, while not modifying the other instances of the same class.

The **Facade pattern** provides a simple and top-level interface for the client and allows it to access the system, without knowing any of the system logic and inner-workings.

Def 2: **Facade pattern** provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

The **Proxy pattern** is used when we want to limit the capabilities and the functionalities of a class, by using another class which limits it by using this proxy class, the client uses the interface it defines, to access the original class. This ensures that the client can't do anything out of order with the original class since all of his requests pass through our proxy class.

The **Flyweight pattern** enables the sharing of objects to support large numbers of fine-grained objects efficiently. A flyweight is a shared object that can be used in multiple contexts simultaneously. The flyweight acts as an independent object in each context.

Behavioral Patterns are concerned about providing solutions regarding *object interaction*
- how do they communicate, how are some dependent on others, and how to segregate them to be both dependent and independent and provide both flexibility and testing capabilities.

The **Chain of Responsibility pattern** is widely used and adopted. It defines a chain of objects that collectively, one after another, process the request - where each processor in the chain has its own processing logic.
Each of these processing units decides who should continue processing the request next, and each has a reference to the next in line. It's important to note that it's very handy for decoupling the sender from the receiver.

Def 2: Chain of responsibility design pattern gives more than one object an opportunity to handle a request by linking receiving objects together in the form of a chain.

The **Command pattern** is useful to abstract the business logic into discrete actions which we call commands. These command objects help in loose coupling between two classes where one class (invoker) shall call a method on another class (receiver) to perform a business operation.

The **Iterator pattern** is used as the core pattern of Java's Collection Framework.Iterator pattern provides a way to access the elements of collection sequentially without exposing its underlying representation.

The **Observer pattern** defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the publish-subscribe pattern.

The **Strategy pattern** is used where we choose a specific implementation of an algorithm or task in run time – out of multiple other implementations for the same task. **Strategy and Decorator are interrelated.**

The **Visitor pattern** is used when we want a hierarchy of objects to modify their behavior but without modifying their source code.

The **State pattern** is used when a specific object needs to change its behavior, based on its state.This is accomplished by providing each of these objects with one or more state objects based on these state objects, we can completely change the behavior of the concerned object. **Visitor and State are interrelated.**

Def 2: In state pattern allows an object to alter its behavior when its internal state changes. The object will appear to change its class. There shall be a separate concrete class per possible state of an object.

The **Template Method**, otherwise known as **Template Pattern** is all around us. It boils down to defining an abstract class that provides predefined ways to run its methods. Sub-classes that inherit these methods must also follow the way defined in the abstract class.

In some cases, the abstract class may already include a method implementation, not just instructions, if it's a functionality that will be shared amongst all or most of the sub-classes.

Def 2: Template method pattern defines the sequential steps to execute a multi-step algorithm and optionally can provide a default implementation as well (based on requirements).

The **Memento pattern** is used to restore the state of an object to a previous state. It is also known as the snapshot pattern.

The **Interpreter pattern** specifies how to evaluate sentences in a language, programatically. It helps in building a grammar for a simple language, so that sentences in the language can be interpreted.

The **Mediator pattern** defines an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets us vary their interaction independently.