Singleton Pattern

What?

This pattern ensures that only one instance/object of a class is created, and it provides global access to that object. Even if it is attempted to create a new instance of the class, the same old instance is returned ensuring creation of only one instance.

Why?

There may be some classes of which one must not use two objects, doing that could lead to major issues. This pattern makes sure that only an object/instance of the class is created and no other object can be instantiated which solves the potential issue that could occur from having more than one instance.

For example :-
Suppose a printer class, whose task is to send data to the printer to print, here creation of two instances of this class can lead to bugs and errors because a printer only prints one thing at a time and if the printer receives instructions from more than one object the printer may not work or may malfunction.

How?

Using the singleton pattern is quite straightforward.

1. We make the default constructor private so that the class cannot be instantiated by ‘new’ keyword.
2. We create a static method to create the instance that calls the private constructor and instantiate the class and store it in a static field. The same stored static instance is returned every time this method is called

Real life use case: Database connections, Loggers etc.

Factory Pattern

What?

Factory pattern is the pattern that allows the client to create related (concrete) products using an abstract interface without knowing or caring about the products that are actually created. This could be defined as an actual factory that creates products for the client without the client having to get involved in the creation process.
A client can have access to multiple factories producing multiple products, based on abstract interfaces.

Why?

This pattern provides dependency inversion, the client being dependent on the abstract interfaces of the factories and the products not caring about the actual factories and products. This decouples the client from any concrete implementations. It makes the creation of concrete products easier and allows for extensions or modifications to the products without affecting the client. Adding more products or factories can also be easily done without messing up the client as the client doesn't care about the concrete implementations.

How?

Factory can create a single product or a family of products

1. We create interfaces for the products and implement them to create concrete products.
2. We create interfaces/abstractions for factories and implement them to create concrete factories, these factories consist of methods to create the products based on the clients need.
3. The client uses a field to store a factory and then tells the factory the products it needs, here the client has no idea of implementation of the factory or the product.

Real life use cases: Simulations, game characters etc.

Builder Pattern

What?

This pattern encapsulates the construction process of a product and allows the product to be constructed in multiple steps in no particular order.

Why?

This pattern is very useful in the classes that have constructors that take a lot of parameters. A number of parameters in a method is generally not a good idea because there must be a specific order of parameters input and maybe we don't even want to provide some parameters. This would lead to constructor telescoping, creating multiple overloaded constructors to fit our needs. Builder pattern solves this issue by breaking down the parameters into multiple methods allowing multistep creation of the product and gives us control over the steps of the construction process.

P.S. This pattern is hugely beneficial for classes with big numbers of constructor parameters, but for small constructors this wouldn't benefit much.

How?

1. We have a class 'ourClass', which we want to create a product of.
2. We create an interface 'productBuilder' with the methods to build the product step by step.
3. We create a concrete 'productBuilder' with a field of type ourClass and methods to add values to the field.
4. The client instantiates productBuilder and provides fields as needed, the productBuilder returns the built product.

Real life use cases: UI components, Database query builders etc.

Decorator Pattern

What?

This is a pattern that allows the user to attach additional features/responsibilities to an object during runtime. This provides a flexible method to extend functionality.

Why?

This pattern helps us to be consistent with the Open/close principle, we can add/extend functionalities without being modified. This also works as an alternative to subclassing/inheriting as this allows us to extend the properties of an object without defining a completely new one. Decorators can modify the behaviour of objects during runtime. Since an object can be wrapped multiple times by different decorator wrappers at runtime it allows for more flexibility than concrete static subclassing.

How?

1. We have an abstract class that defines our components.
2. This is implemented to create a concrete component, this is the component that will be wrapped by the decorator.
3. We now implement the same abstract class to an abstract decorator, this decorator also contains a field to hold an instance of the abstract class. Basically the decorator has both ‘is-a’ and ‘has-a’ relationship with the main component.
4. We now create concrete decorators with implemented methods and other different behaviours. These concrete decorators take an instance of the main component and wrap it with additional behaviours.

Real life use cases: Game character powerups, Text formatters etc.

Facade Pattern

What?

This is the pattern that provides a simple interface to interact with a complex set of classes and methods. It provides an abstraction to the complex interactions behind a simple interface.

Why?

A system may have many components that need to work together. Without a facade the user has to do everything manually which can be complicated and not efficient. What facade does is it provides a smooth and simple interface to the user and does everything behind it.

For Example:-
A computer may have a startComputer() method but behind that method internally it calls loadCPU(), checkRAM(), bootOS() etc. startComputer is the facade here. Providing us a simple interface to start the computer easily without us even knowing the internal implementation.

P.s Facade doesn't prevent clients from using individual components, they still have control over individual components.

How?

This is also a quite straightforward pattern

1. Create/Have class(es) and methods in them.
2. Create a facade class with instance fields of the classes we are creating this facade for.
3. Create facade methods that combine and call different methods from the classes we have in instance fields.
4. The client can now directly call the facade instead of calling all the different methods manually.

Real life use cases: Theatre system, Payment system etc.

Observer Pattern

What?

This is a pattern which defines a mechanism that allows multiple objects to automatically get notified about any changes in the object they're observing.

Why?

This is generally useful in event handling conditions where when one thing changes other things need to react to that change. The objects can be loosely connected so they don't know each other's details but still can share those changes. There is one-to-many dependency between subjects and observers, when subject changes all of its dependents are notified.

For Example:
A store with a system that notifies its customers when a new collection of clothes arrive. The customers are observing the store and when the new collection arrives the system automatically notifies the customers via email or other media.

How?

1. Create interfaces for Observers, Subjects. We can have multiple Observers and subjects. An observer itself also can be a subject that others observers can observe.
2. Create concrete implementations of the Subjects and Observers.
3. Subjects hold a list of all of its observers and methods like registerObserver, notifyObservers, removeObserver etc.
4. When anything is changed, notifyObservers is called and it uses the list of observers to notify them
5. Observers have other methods and an update method which is called by the subject which updates the observers with the updated data.

Real life use cases: Bank notification system, YouTube Notification system, Weather monitoring system etc.

Iterator Pattern

What?

This is a pattern that allows clients to iterate over elements of a collection or an iterable object without exposing the internal representation. The client can iterate over the collection without even knowing how the data is stored or structured, achieving decoupling.

Why?

Collections can be structured and stored in various forms. Collections must provide a way to iterate over the elements for other code to use its elements. When we have to use multiple collections with different ways of storing data, accessing them gets very complicated. We will need to couple every collection with our client which can cause multiple issues.

Iterator pattern allows collections to be based on interface and also provide an iterator that can iterate over its elements, the client can now stay decoupled from the collection and still iterate and access its data regardless of the structure of the store data. We can also use different iterating methods, just by creating a new iterator class.

How?

1. Create an interface for the iterator with methods to iterate over a collection.(hasNext(), hasPrevious(), Next(), Previous() etc.)
2. Create concrete iterators implementing the interface that can iterate over a specific structure. Iterators must implement the same interface so that the client code is compatible with any collection type.
3. Create collections with elements and a method that provides an iterator that can iterate over its data.
4. Now, the client can just ask the collection to provide the iterator and can use it easily, without knowing or caring about the type of collection. (All collections now return the same type of iterator implemented from the same interface)

Real life use cases: File browsers, Music players etc.