Creational Design Patterns

1. **Singleton Pattern**

Ensures that at most only one instance of an object exists throughout application.

This is a widely used approach for a Singleton class as it doesn’t require synchronization, is thread safe, enforces lazy initialization.

Also, note that the constructor has the private access modifier. This is a requirement for creating a Singleton since a public constructor would mean anyone could access it and start creating new instances.

**When to use:**

* For resources that are expensive to create (like **database connection objects**)
* It’s good practice to keep all **loggers** as Singletons which increases performance.
* Classes which provide access to **configuration settings** for the application
* Classes that contain **resources** that are accessed in **shared** mode

1. **Abstract Factory Pattern**

which provides an interface for creating related or dependent objects without specifying the objects' concrete classes.

The Abstract Factory Design Pattern is used to create families of related or dependent objects. It’s also sometimes called a **factory of factories**. (contrast to Factory method pattern).

**When to use:**

* The client is **independent of how we create** and compose the objects in the system
* The system consists of **multiple families of objects**, and these families are designed to be used together
* We **need a run-time value to construct** a particular dependency

1. **Factory Method Pattern**

Creates objects of several related classes without specifying the exact object to be created.

The Factory Design Pattern or Factory Method Design Pattern is one of the most used design patterns. This pattern “defines an interface for creating an object, but let subclasses decide which class to instantiate.

This pattern delegates the responsibility of initializing a class from the client to a particular factory class. To achieve this, we rely on a factory which provides us with the objects, hiding the actual implementation details.

Example of Polygon get the class of different polygons by specifying number of sides.

**When to use:**

* When the implementation of an interface or an abstract class is **expected to change frequently**
* When the current implementation **cannot** comfortably **accommodate new change**
* When the initialization process is relatively simple, and the constructor only **requires a handful of parameters**

1. **Builder Pattern**

which separates the construction of a complex object from its representation so that the same construction process can create different representations.

When the **complexity of creating object increases**, the Builder pattern can separate out the instantiation process by using another object (a builder) to construct the object.

This builder can then be used to create many other similar representations using a simple step-by-step approach.

For example, HostBuilder in .Net Core program.cs file (we have created HostingBuilder)

**When to use:**

* When the process involved in creating an **object is extremely complex**, with lots of mandatory and optional parameters
* When an increase in the **number of constructor parameters leads to a large list** of constructors
* When **client expects different representations** for the object that’s constructed