Summary

Summary

Pattern	Purpose
Builder Pattern	The builder pattern is used to create objects. It seperates out how the object is represented and how it is created. Additionally, it breaks down the creation into multiple steps. For instance in Java the `java.lang.StringBuilder` is an example of the builder pattern.
Singleton Pattern	The singleton pattern is applied to restrict instantiation of a class to only one instance. For instance in the Java language the class `java.lang.Runtime` is a singleton.

Prototype Pattern	Prototype pattern involves creating new objects by copying existing objects. The object whose copies are made is called the **prototype**. In Java the `clone()` method of `java.lang.Object` is an example of this pattern.
Factory Method Pattern	The factory method is defined as providing an interface for object creation but delegating the actual instantiation of objects to subclasses. For instance the method `getInstance()` of the class `java.util.Calendar` is an example of a factory method pattern.
Abstract Factory	The abstract factory pattern is defined as defining an interface to create families of related or dependent objects without specifying their concrete classes.The abstract factory is particularly useful for frameworks and toolkits that work on different operating systems. For instance, if your library provides fancy widgets for the UI, then you may need a family of products that work on MacOS and a similar family of products that work on Windows.

Adapter Pattern	The Adapter pattern allows two incompatible classes to interoperate that otherwise can't work with eachother. Consider the method `asList()` offered by `java.util.Arrays` as an exampe of the adapter pattern. It takes an array and returns a list.
Bridge Pattern	The bridge pattern describes how to pull apart two software layers fused together in a single class hierarchy and change them into parallel class hierarchies connected by a bridge
Composite Pattern	The pattern allows you to treat the whole and the individual parts as one. The closest analogy you can imagine is a tree. The tree is a recursive data-structure where each part itself is a sub-tree except for the leaf nodes.

Decorator Pattern	The decorator pattern can be thought of as a wrapper or more formally a way to enhance or extend the behavior of an object dynamically. The pattern provides an alternative to subclassing when new functionality is desired. A prominent example of this pattern is the `java.io` package, which includes several decorators. For example the `BufferedInputStream` wraps the `FileInputStream` to provide buffering capabilities.
Facade Pattern	The facade pattern is defined as a single uber interface to one or more subsystems or interfaces intending to make use of the subsystems easier
Flyweight Pattern	The pattern advocates reusing state among a large number of fine grained object. Methods `java.lang.Boolean.valueOf()` and `java.lang.Integer.valueOf()` both return flyweight objects.

Proxy Pattern	In a proxy pattern setup, a proxy is responsible for representing another object called the subject in front of clients. The real subject is shielded from interacting directly with the clients. The `java.rmi.*` package contains classes for creating proxies. RMI is Remote Method Invocation. It is a mechanism that enables an object on one Java virtual machine to invoke methods on an object in another Java virtual machine.
Chain of Responsibility Pattern	In a chain of responsibility pattern implementation, the sender's request is passed down a series of handler objects till one of those objects, handles the request or it remains unhandled and falls off the chain. Multiple objects are given a chance to handle the request. This allows us to decouple the sender and the receiver of a request. The `log()` method of the `java.util.logging.Logger` class is an example of this pattern.

Observer Pattern (Publisher/Subscriber)	The pattern is formally defined as a one to many dependency between objects so that when one object changes state all the dependents are notified. All types implementing the interface `java.util.EventListener` are examples of this pattern.
Interpreter Pattern	The interpreter pattern converts a language's sentences into its grammar and interprets them.
Command Pattern	The pattern is defined as representing an action or a request as an object that can then be passed to other objects as parameters, allowing parameterization of clients with requests or actions. The requests can be queued for later execution or logged. Logging requests enables undo operations. Types implementing the interface `java.lang.Runnable` are examples of this pattern.

Iterator Pattern	An iterator is formally defined as a pattern that allows traversing the elements of an aggregate or a collection sequentially without exposing the underlying implementation. All types implementing the `java.util.Iterator` interface are examples of this pattern.
Mediator Pattern	The pattern is applied to encapsulate or centralize the interactions amongst a number of objects. Object orientated design may result in behavior being distributed among several classes and lead to too many connections among objects. The encapsulation keeps the objects from referring to each other directly and the objects don't hold references to each other anymore. The `java.util.Timer` class represents this pattern where tasks may be scheduled for one-time execution, or for repeated execution at regular intervals in a background thread.

Memento Pattern	The memento pattern let's us capture the internal state of an object without exposing its internal structure so that the object can be restored to this state later. Classes implementing `java.io.Serializable` interface are examples of the memento pattern.
State Pattern	The state pattern encapsulates the various states a machine can be in. The machine or the context, as it is called in pattern-speak, can have actions taken on it that propel it into different states. Without the use of the pattern, the code becomes inflexible and littered with if-else conditionals.
Template Method	The template method pattern defines the skeleton or steps of an algorithm but leaves opportunities for subclasses to override some of the steps with their own implementations. Non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap` are examples of this pattern.

Strategy Pattern	The pattern allows grouping related algorithms under an abstraction, which the client codes against. The abstraction allows switching out one algorithm or policy for another without modifying the client. `java.util.Comparator` has the method `compare()` which allows the user to define the algorithm or strategy to compare two objects of the same type.
Visitor Pattern	The visitor pattern allows us to define an operation for a class or a class hierarchy without changing the classes of the elements on which the operation is performed. The pattern is suitable in scenarios, where the object structure class or the classes that make up its elements don't change often but new operations over the object structure are desired. `java.nio.file.FileVisitor` interface has an implementation class of `SimpleFileVisitor` which is an example of a visitor. The interface is defined as a visitor of files. An implementation of this interface is provided to the `Files.walkFileTree()` methods to visit each file in a file tree.