Creational design patterns - Singleton and Builder

Key terms

Design patterns

> A design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system.

Creational design patterns

> Creational design patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

Singleton

> The singleton pattern is a software design pattern that restricts the instantiation of a class to one object. This is useful when exactly one object is needed to coordinate actions across the system.

Builder

> Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Singleton

Problem

- `Shared resource` - Imagine you have a class that is responsible for managing the database connection. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple database connections, which is not what you want. Similarly, there can be a class that is responsible for managing the logging mechanism. You want to make sure that only one instance of this class exists in your application. If you create multiple instances of this class, you will end up with multiple log files, which is not what you want.
- `Single access point` - Applications often require configuration. For example, you might want to configure the database connection parameters. You want to make sure that only one instance of this class exists in your application. A configuration class should have a single access point to the configuration parameters. If you create multiple instances of this class, you will end up with multiple configuration files.

Solution

Singleton pattern is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance. To implement the Singleton patter, the following steps are required:

- `Constructor hiding` - The constructor of the singleton class should be private or protected. This will prevent other classes from instantiating the singleton class.
- `Global access point` - The singleton class should provide a global access point to get the instance of the singleton class. This global access point should be static and should return the same instance of the singleton class every time it is called. If the instance does not exist, it should create the instance and then return it.

Simple singleton

The first step is to hide the constructor by making it private. This will prevent other classes from instantiating the singleton class.

```
public class Database {
    private Database() {
    }
}
```

The above code restricts the instantiation of the Database class. Now, we need to provide a global access point to get the instance of the Database class. We can do this by creating a static method that returns the instance of the Database class. If the instance does not exist, it should create the instance and then return it.

```
public class Database {
    private static Database instance = new Database();

    private Database() {
    }

    public static Database getInstance() {
        return instance;
    }
}
```

To implement the getInstance() method, we need to create a static variable of the Database class. This variable will hold the instance of the Database class. We will initialize this variable to null. The getInstance() method will check if the instance variable is null. If it is null, it will create a new instance of the Database class and assign it to the instance variable. Finally, it will return the instance variable. This is known as lazy initialization.

```
public class Database {
    private static Database instance = null;

    private Database() {
    }

    public static Database getInstance() {
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }
}
```

Thread-safe singleton

The above code is not thread-safe. If two threads call the getInstance() method at the same time, both threads will check if the instance variable is null. Both threads will find that the instance variable is null. Both threads will create a new instance of the Database class. This will result in two instances of the Database class. To make the above code thread-safe, we can make the getInstance() method synchronized.

```
public class Database {
    private static Database instance = null;

    private Database() {
    }

    public static synchronized Database getInstance() {
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }
}
```

Double-checked locking

The above code is thread-safe. However, it is not efficient. If two threads call the getInstance() method at the same time, both threads will check if the instance variable is null. Both threads will find that the instance variable is null. Both threads will wait for the lock to be released. Once the lock is released, one thread will create a new instance of the Database class. The other thread will wait for the lock to be released. Once the lock is released, it will create a new instance of the Database class. This will result in two instances of the Database class. To make the above code efficient, we can use double-checked locking.

```
public class Database {
    private static Database instance = null;

    private Database() {
    }

    public static Database getInstance() {
        if (instance == null) {
            synchronized (Database.class) {
                if (instance == null) {
                    instance = new Database();
                }
            }
        }
        return instance;
    }
}
```

Summary

- The singleton pattern is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.
- Use cases of singleton pattern
    - Shared resource like database connection, logging mechanism

    - Object that should be instantiated only once like configuration object
- Hide the constructor of the singleton class by making it private so that other classes cannot instantiate the singleton class.
- Add a static method that returns the instance of the singleton class. If the instance does not exist, it should create the instance and then return it.
- Thread safety
    - Make the `getInstance()` method synchronized.
    - Use double-checked locking.

Builder

Problems

- `Complex object creation` - There are multiple ways to create an object, but constructors are the primary technique used for creating instances of a class. However, constructors become unmanageable when there is a need to create an object with many parameters. This is known as the telescoping constructor anti-pattern. The telescoping constructor anti-pattern is a code smell that indicates that the class has too many constructors. This is a code smell because it is difficult to maintain and extend the class.
- `Validation and failing object creation` - There are cases when you want to validate the parameters before creating an object. For example, you might want to validate the parameters before creating a database connection. If the parameters are invalid, you might want to throw an exception. However, if we use the default constructor, we cannot fail object creation.
- `Immutability` - Mutable objects are objects whose state can be changed after they are created. Immutable objects are objects whose state cannot be changed after they are created. Immutable objects are easier to maintain and extend whereas mutable objects can lead to bugs. However, if we use the default constructor, we cannot create immutable objects.

Constructor with a hash map

The above problems can be solved using a constructor with a hash map. The constructor will take a hash map as a parameter. The hash map will contain the parameters and their values. The constructor will validate the parameters and create the object.

```
public class Database {
    private String host;
    private int port;
    private String username;
    private String password;

    public Database(Map<String, String> config) {
        if (config.containsKey("host")) {
            this.host = config.get("host");
        }
        if (config.containsKey("port")) {
            this.port = Integer.parseInt(config.get("port"));
        }
        if (config.containsKey("username")) {
            this.username = config.get("username");
        }
        if (config.containsKey("password")) {
            this.password = config.get("password");
        }
    }
}
```

Some problems with the above code are:

- `Type safety` - A hash map cannot have values with different types. If we want to use different types, we need to use a hash map with a string key and an object value. However, this will result in a runtime error if we try to cast the object to the wrong type.
- `Defined parameters` - With the above approach, identifying the parameters is difficult. We need to read the code to identify the parameters. This is not a good approach because it is difficult to maintain and extend the code.

Inner class

Instead of using a hash map, we can use a class to accept parameters for object creation. The parameter class is type safe, and it is easy to identify the parameters.

```
public class Database {
    private String host;
    private int port;
    private String username;
    private String password;

    public Database(DatabaseParameters parameter) {
        this.host = parameter.host;
        this.port = parameter.port;
        this.username = parameter.username;
        this.password = parameter.password;
    }
}

class DatabaseParameters {
    public String host;
    public int port;
    public String username;
    public String password;
}
```

The above code is type safe. However, it is not easy to use. We need to create an instance of the DatabaseParameters class and then pass it to the Database class. This is not a good approach because it is difficult to maintain and extend the code. Similarly, if we even want to change a single parameter name, we have to open the database class for modification. Instead, we should move the destructuring of the parameter class and validation logic to the Parameter class. This will require creating a Database constructor with all the fields. Again, why would developers not just want to use the constructor instead?
So we need a way to allow the parameter class to create the Database object while not exposing a constructor. This can be done using an inner class. This inner class is known as the builder class.

```
public class Database {
    private String host;
    private int port;
    private String username;
    private String password;

    private Database() {
    }

    public static class DatabaseBuilder {
        private String host;
        private int port;
        private String username;
        private String password;

        public Database build() {
            Database database = new Database();
            database.host = this.host;
            database.port = this.port;
            database.username = this.username;
            database.password = this.password;
            return database;
        }
    }
}
```

The above code now allows us to create a Database object using the DatabaseBuilder class. We can fail object creation by adding a validation hook to the build method. The objects created are immutable because the Database class does not have any setters. And the developer can create objects with any permutation of parameters.

```
Database database = new Database.DatabaseBuilder()
    .host("localhost")
    .port(3306)
    .username("root")
    .password("password")
    .build();
```

Summary

- The builder pattern is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
- Use cases of builder pattern
    - Complex object creation - Telescoping constructor anti-pattern
    - Validation and failing object creation
    - Immutability
- Add a static inner class to the class that you want to create. This inner class is known as the builder class.

- Add a private constructor to the class that you want to create. This constructor will be used by the builder class to create the object.
- Implement the `build()` method in the builder class. This method will return the object created by the private constructor.
- Add a method for each parameter in the builder class. This method will set the parameter value and return the builder class instance.

Reading list

- Telescoping constructor anti-pattern (https://www.vojtechruzicka.com/avoid-telescoping-constructor-pattern/)
- Why objects should be immutable? (https://octoperf.com/blog/2016/04/07/why-objects-must-be-immutable)