**Builder Pattern**

**Builder Pattern**

Builder Pattern is a design pattern that allows step by step creation of a complex object. Builder design pattern addresses the below points

* If there are too many arguments to pass from the program to the class that can be error prone because most of the time, the type of arguments are the same and from the client side it's hard to maintain the order of the argument.
* Some of the parameters might be optional, but we are forced to send all the parameters and optional parameters need to send as NULL.

Let's understand with an example. Book Class as shown below one constructor taking all the parameters to be set, and the corresponding getters to read the object’s fields once the object has been created. This class also has two mandatory parameters and 4 optional parameters.

import builder.pattern.Genre;

import java.time.Year;

public class Book {

private final String isbn;// mandatory parameter

private final String title;// mandatory parameter

private final Genre genre;//optional parameter

private final String author;//optional parameter

private final Year published;//optional parameter

private final String description;//optional parameter

public Book(String isbn, String title, Genre genre, String author, Year published, String description) {

this.isbn = isbn;

this.title = title;

this.genre = genre;

this.author = author;

this.published = published;

this.description = description;

}

public String getIsbn() { return isbn;}

public String getTitle() {return title; }

public Genre getGenre() {return genre; }

public String getAuthor() {return author; }

public Year getPublished() {return published;}

public String getDescription() {return description; }

}

Any calling application which creates an object from the above class must send arguments in proper order and the calling application can get prone to error as most of the arguments are the same data type. Some programming languages support named parameters which might help but not all languages support named parameters.

We have a constructor which expects to pass all parameters including mandatory and optional. If the optional parameter is not present, the calling application must send a null value in the constructor or the class need to have a constructor for each given combination.

We can solve the above issues using the builder pattern.

**Implementation**

To implement a singleton pattern, Class should have the below properties

* First of all you need to create a [static nested class](https://www.digitalocean.com/community/tutorials/java-inner-class) and then copy all the arguments from the outer class to the Builder class. We should follow the naming convention and if the class name is Book then builder class should be named as BookBuilder.
* Java Builder class should have a public constructor with all the required attributes as parameters.

Java Builder class should have methods to set the optional parameters and it should return the same Builder object after setting the optional attribute.

* The final step is to provide a build() method in the builder class that will return the Object needed by client program. For this we need to have a private constructor in the Class with Builder class as argument.

Example:

import builder.pattern.Genre;

import java.time.Year;

public class Book {

private final String isbn;// mandatory parameter

private final String title;// mandatory parameter

private final Genre genre;//optional parameter

private final String author;//optional parameter

private final Year published;//optional parameter

private final String description;//optional parameter

public Book(BookBuilder builder ) {

this.isbn = builder.isbn;

this.title = builder.title;

this.genre = builder.genre;

this.author = builder.author;

this.published = builder.published;

this.description = builder.description;

}

public String getIsbn() { return isbn;}

public String getTitle() {return title; }

public Genre getGenre() {return genre; }

public String getAuthor() {return author; }

public Year getPublished() {return published;}

public String getDescription() {return description; }

public static class BookBuilder {

private final String isbn;

private final String title;

private Genre genre;

private String author;

private Year published;

private String description;

public BookBuilder (String isbn, String title) {

this.isbn = isbn;

this.title = title;

}

public BookBuilder genre(Genre genre) {

this.genre = genre;

return this;

}

public BookBuilder author(String author) {

this.author = author;

return this;

}

public BookBuilder published(Year published) {

this.published = published;

return this;

}

public BookBuilder description(String description) {

this.description = description;

return this;

}

public BookBuilder build() {

return new Book(this);

}

}

}

**Key Points**

* The scope of the Book constructor has been changed to private, so that it cannot be accessed from the outside of the Book class. This makes it impossible to create a Book instance directly. The object creation process is delegated to the Builder class.
* The Book constructor takes a Builder instance as its only parameter, which contains all the values to be set by the Book constructor.
* The Builder class contains the same fields as the Book class, which is necessary to hold the values to be passed to the Book constructor. This has often been rightly criticized as code duplication.
* For every optional field to be set, the Builder class exposes a setter-like method, which assigns the field’s value and returns the current Builder instance to build the object in a fluent way.
* The build method calls the Book constructor by passing the current Builder instance as the only parameter. The values held by the Builder instance are then unpacked by the Book constructor, which assigns them to the corresponding Book fields.