Builder Pattern in java

**Method Chaining:**

In java, Method Chaining is used to invoke multiple methods on the same object which occurs as a single statement. Method-chaining is implemented by a series of methods that return the [this](https://www.geeksforgeeks.org/this-reference-in-java/) reference for a class instance.

**Example :**

Coffee coffee =

new Coffee.Builder("Mocha").milktrue).sugar(false).size("Large").build();

**Implementation:**

As return values of methods in a chain is [this](https://www.geeksforgeeks.org/this-reference-in-java/) reference, this implementation allows us to invoke methods in chain by having the next method invocation on the return value of the previous method in the chain.

// Java code to demonstrate method chaining

final class Student {

// instance fields

private int id;

private String name;

private String address;

// Setter Methods

// Note that all setters method

// return this reference

public Student setId(int id)

{

this.id = id;

return this;

}

public Student setName(String name)

{

this.name = name;

return this;

}

public Student setAddress(String address)

{

this.address = address;

return this;

}

@Override

public String toString()

{

return "id = " + this.id + ", name = " + this.name +

", address = " + this.address;

}

}

// Driver class

public class MethodChainingDemo {

public static void main(String args[])

{

Student student1 = new Student();

Student student2 = new Student();

student1.setId(1).setName("Ram").setAddress("Noida");

student2.setId(2).setName("Shyam").setAddress("Delhi");

System.out.println(student1);

System.out.println(student2);

}

}

Output:

id = 1, name = Ram, address = Noida

id = 2, name = Shyam, address = Delhi

**Need of Builder Pattern :**Method chaining is a useful design pattern but however if **accessed concurrently**, a thread may observe some fields to contain inconsistent values. Although all setter methods in above example are *atomic*, but calls in the method chaining can lead to inconsistent object state when the object is modified concurrently. The below example can lead us to a *Student* instance in an **inconsistent state**, for example, a student with name *Ram* and address *Delhi*.

// Java code to demonstrate need of Builder Pattern

// Server Side Code

final class Student {

// instance fields

private int id;

private String name;

private String address;

// Setter Methods

// Note that all setters method

// return this reference

public Student setId(int id)

{

this.id = id;

return this;

}

public Student setName(String name)

{

this.name = name;

return this;

}

public Student setAddress(String address)

{

this.address = address;

return this;

}

@Override

public String toString()

{

return "id = " + this.id + ", name = " + this.name +

", address = " + this.address;

}

}

// Client Side Code

class StudentReceiver {

private final Student student = new Student();

public StudentReceiver()

{

Thread t1 = new Thread(new Runnable() {

@Override

public void run()

{

student.setId(1).setName("Ram").setAddress("Noida");

}

});

Thread t2 = new Thread(new Runnable() {

@Override

public void run()

{

student.setId(2).setName("Shyam").setAddress("Delhi");

}

});

t1.start();

t2.start();

}

public Student getStudent()

{

return student;

}

}

// Driver class

public class BuilderNeedDemo {

public static void main(String args[])

{

StudentReceiver sr = new StudentReceiver();

System.out.println(sr.getStudent());

}

}

Output may be:

id = 2, name = Shyam, address = Noida

id = 0, name = null, address = null

**Note :**Try running *main* method statements in loop(i.e. multiple requests to server simultaneously).

To solve this problem, there is Builder pattern to ensure the **thread-safety** and **atomicity** of object creation.

**Implementation :**In Builder pattern, we have a [inner static class](https://www.geeksforgeeks.org/nested-classes-java/) named *Builder* inside our Server class with instance fields for that class and also have a [factory method](https://www.geeksforgeeks.org/design-patterns-set-2-factory-method/) to return an **new** instance of *Builder*class on every invocation.

The setter methods will now return *Builder* class reference. We will also have a *build* method to return instances of Server side class, i.e. outer class.

// Java code to demonstrate Builder Pattern

// Server Side Code

final class Student {

// final instance fields

private final int id;

private final String name;

private final String address;

public Student(Builder builder)

{

this.id = builder.id;

this.name = builder.name;

this.address = builder.address;

}

// Static class Builder

public static class Builder {

/// instance fields

private int id;

private String name;

private String address;

public static Builder newInstance()

{

return new Builder();

}

private Builder() {}

// Setter methods

public Builder setId(int id)

{

this.id = id;

return this;

}

public Builder setName(String name)

{

this.name = name;

return this;

}

public Builder setAddress(String address)

{

this.address = address;

return this;

}

// build method to deal with outer class

// to return outer instance

public Student build()

{

return new Student(this);

}

}

@Override

public String toString()

{

return "id = " + this.id + ", name = " + this.name +

", address = " + this.address;

}

}

// Client Side Code

class StudentReceiver {

// volatile student instance to ensure visibility

// of shared reference to immutable objects

private volatile Student student;

public StudentReceiver()

{

Thread t1 = new Thread(new Runnable() {

@Override

public void run()

{

student = Student.Builder.newInstance()

.setId(1)

.setName("Ram")

.setAddress("Noida")

.build();

}

});

Thread t2 = new Thread(new Runnable() {

@Override

public void run()

{

student = Student.Builder.newInstance()

.setId(2)

.setName("Shyam")

.setAddress("Delhi")

.build();

}

});

t1.start();

t2.start();

}

public Student getStudent()

{

return student;

}

}

// Driver class

public class BuilderDemo {

public static void main(String args[])

{

StudentReceiver sr = new StudentReceiver();

System.out.println(sr.getStudent());

}

}

Output is guaranteed to be one of below:

id = 1, name = Ram, address = Noida

OR

id = 2, name = Shyam, address = Delhi

The *Builder.newInstance()* factory method can also be called with any required arguments to obtain a *Builder* instance by overloading it. The object of *Student* class is constructed with the invocation of the *build()* method.  
The above implementation of Builder pattern makes the *Student* class **immutable** and consequently **thread-safe**.

Also note that the *student* field in client side code cannot be declared [final](https://www.geeksforgeeks.org/final-keyword-java/) because it is assigned a new immutable object. But it be declared [volatile](https://www.geeksforgeeks.org/volatile-keyword-in-java/) to ensure visibility of shared reference to immutable objects. Also private members of *Builder* class maintain encapsulation.

Please have a look at [append](https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/lang/StringBuilder.java) method of [*StringBuilder*](https://contribute.geeksforgeeks.org/conversion-between-types-of-strings-in-java-combined-with-httpwww-geeksforgeeks-orgg-fact-27-string-vs-stringbuilder-vs-stringbuffer/) class in *java.lang* package to understand implementations of Builder pattern more.