Builder pattern

A creational pattern
Building a complex object

Let’s say that we have a class for pretty complex objects, with lots of internal instance variables.

We could provide a lot of constructors to allow for flexibility, but the problem is that some of the parameters might be optional and only a few mandatory.

How should we write the parameter lists to fit as many combinations as possible? And what if many of the parameters are of the same type?
public class Reservation{
    public static final int FIRST_CLASS=0;
    public static final int SECOND_CLASS=1;
    public static final int ECONOMY_CLASS=2;
    // Mandatory fields
    private int seatClass;
    private int numSeats;
    private Date day;
    // Optional fields
    private int numMeals;
    private boolean isWindowSeat;
    private boolean isNonSmoking;
    private boolean hasTable;

    public Reservation(Date day, int seatClass, int numSeats){
        this.day = day;
        this.seatClass = seatClass;
        this.numSeats = numSeats;
    }
    // How to overload this constructor with combinations of
    // ints and booleans? It is not practical!
}
One use of the builder pattern

It is possible to apply the builder pattern to solve this situation.

A builder encapsulates the construction of an object and also allows for it to be constructed in steps (in any order).
Reservation builder

Create a nested static class inside the Reservation class and call it Builder:

```java
public class Reservation{
    public static final int FIRST_CLASS=0;
    public static final int SECOND_CLASS=1;
    public static final int ECONOMY_CLASS=2;
    // more fields

    public static class Builder{
        // same instance variables as Reservation
        public Builder(Date day, int seatClass, int numSeats){
            // set these mandatory fields
        }
        // Methods for setting optional fields
    }
    // more stuff in the Reservation class
}
```
Builder

Duplicate all fields from the enclosing class (in this case Reservation)

Make a constructor which takes the few mandatory fields

Make methods for setting optional fields which return this (of type Builder)

Make a build() method which returns the product to build (Reservation)

Make the constructor of Reservation private and accept a builder
Duplicate all fields from the enclosing class (in this case Reservation)

public static class Builder{
  //Mandatory
  private int seatClass;
  private int numSeats;
  private Date day;
  // Optional fields
  private int numMeals;
  private boolean isWindowSeat;
  private boolean isNonSmoking;
  private boolean hasTable;
  ...
}
Builder

Make a constructor which takes the few mandatory fields

```java
public Builder(Date day, int seatClass, int numSeats){
    this.day = day;
    this.seatClass = seatClass;
    this.numSeats = numSeats;
}
```
Builder

Make methods for setting optional fields which return this (of type Builder)

```java
public Builder numMeals(int numMeals) {
    this.numMeals = numMeals; return this;
}
public Builder isWindowSeat(boolean isWindowSeat) {
    this.isWindowSeat = isWindowSeat; return this;
}
public Builder isNonSmoking(boolean isNonSmoking) {
    this.isNonSmoking = isNonSmoking; return this;
}
public Builder hasTable(boolean hasTable) {
    this.hasTable = hasTable; return this;
}
```
Builder

Make a build() method which returns the product to build (Reservation)

```java
public Reservation build(){
    return new Reservation(this);
}
```
Builder

Make the constructor of Reservation private and accept a builder and copy all fields:

```java
// in the outer class Reservation!
private Reservation(Builder b){
    this.day = b.day;
    this.seatClass = b.seatClass;
    this.numSeats = b.numSeats;
    this.isWindowSeat = b.isWindowSeat;
    this.isNonSmoking = b.isNonSmoking;
    this.hasTable = b.hasTable;
}
```
How to use the Builder to create a Reservation

```java
import java.util.Date;
import java.text.SimpleDateFormat;
public class Main{
    public static void main(String[] args){
        Date day = new Date();
        try{
            day = new SimpleDateFormat("yyyy-MM-dd").parse("2020-02-28");
        }catch(Exception e){}
        Reservation res = new Reservation.Builder(day, Reservation.FIRST_CLASS, 2)
            .numMeals(2)
            .isNonSmoking(true)
            .isWindowSeat(true)
            .build();
        System.out.println(res);
    }
}
$ javac Main.java && java Main
Trip on Fri Feb 28 00:00:00 CET 2020 in First class for 2 people with 2 meals window seat
Non-smoking
```
Imagine creating a Reservation without a Builder

```java
import java.util.Date;
import java.text.SimpleDateFormat;
public class Main{
    public static void main(String[] args){
        Date day = new Date();
        try{
            day = new SimpleDateFormat("yyyy-MM-dd").parse("2020-02-28");
        }catch(Exception e){}
        Reservation res = new Reservation
            (day, Reservation.FIRST_CLASS, 2, 2, true, true, false);
        // Who will remember what that means? What if we flip some parameters by mistake?
        // Who will remember the correct order of the arguments?
        System.out.println(res);
    }
}
$ javac Main.java && java Main
Trip on Fri Feb 28 00:00:00 CET 2020 in First class for 2 people with 2 meals window seat
Non-smoking
Recap

Make a static inner class Builder with the exact same instance variables

Make a constructor for Builder take the mandatory arguments

Add methods for optional arguments who sets them and returns this

Add a method build() which returns the product being built, by calling the products constructor (which is private and takes a builder as argument)

In the products constructor (e.g. Reservation), copy all fields from the builder to the product.
How can the weird syntax work?

```java
Reservation res = new Reservation.Builder
    (day, Reservation.FIRST_CLASS, 2)
    .numMeals(2)
    .isNonSmoking(true)
    .isWindowSeat(true)
    .build();
```

`numMeals()` returns a builder, so we can add a call to `isNonSmoking()` which returns a builder so we can add call to `isWindowSeat()` which return a builder so we can add call to `build()` which returns a reference to a Reservation.
Order doesn’t matter

We can create another reservation like this:

```java
Reservation res = new Reservation.Builder
  (day, Reservation.FIRST_CLASS, 2)
  .numMeals(2)
  .isNonSmoking(true)
  .isWindowSeat(true)
  .build();
System.out.println(res);
Reservation other =
  new Reservation.Builder(day, Reservation.ECONOMY_CLASS, 1)
  .isWindowSeat(true)
  .numMeals(4)
  .isNonSmoking(true)
  .hasTable(true)
  .build();
System.out.println(other);
```

Trip on Fri Feb 28 00:00:00 CET 2020 in First class for 2 people with 2 meals window seat
Non-smoking
Trip on Fri Feb 28 00:00:00 CET 2020 in Economy class for 1 people with 4 meals window seat
Non-smoking with a table