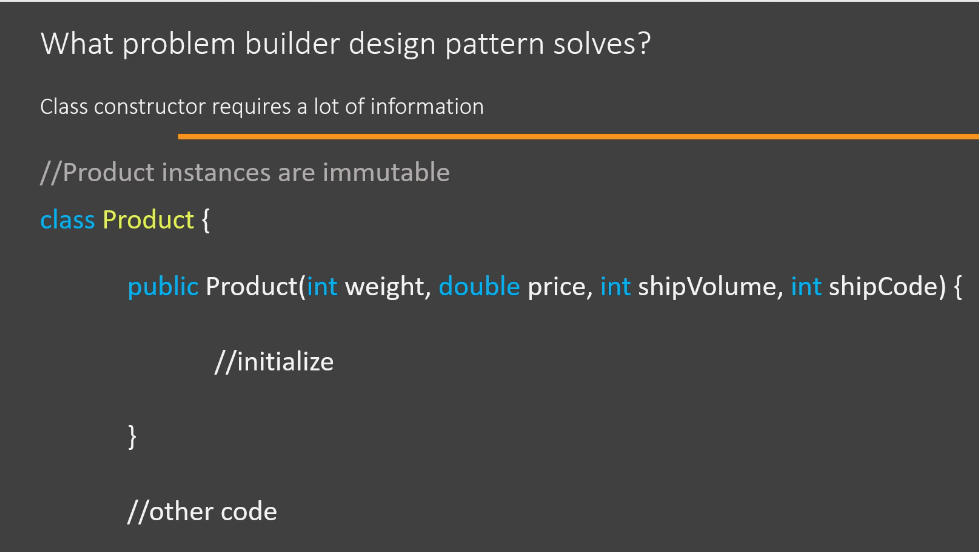
Builder design pattern

a) Introduction

Now, this is a creational design pattern. That means we are going to use builder when we are creating objects of a class. But before we go into details of this design pattern, let's first look at some of the problems that this design pattern tries to solve.



What problem builder design pattern solves? --> Constructor of a class requires a lot of information

//Product instances are immutable

class Product {

public Product(int weight, double price, int shipVolume, int shipCode) {

//initialize

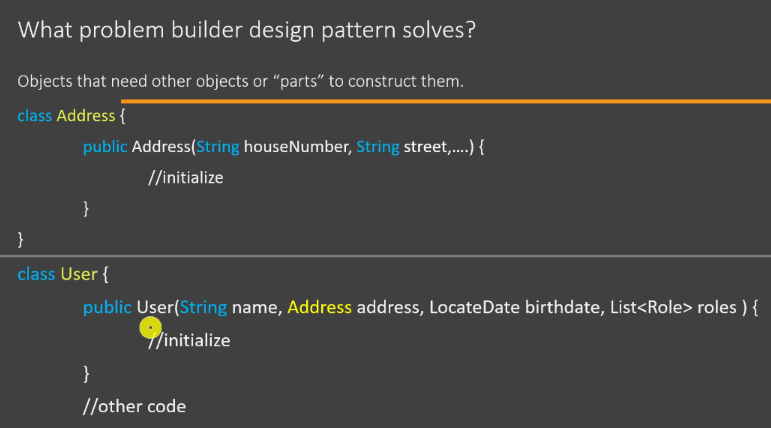
}

//other code

}

So here I have a simple class called as 'Product', and let's say we have a requirement that objects of this class should be immutable. Now, an immutable object is an object whose state or properties cannot be changed once that object is created. String class is a good example. It is an inbuilt Java class and objects of String class are immutable. So, once you create a String object, you cannot change its value. When you are writing immutable classes, you will often end up with constructors with multiple parameters because you have to provide all the state information for the object in the constructor itself. Now, having a method or a constructor that needs a lot of arguments is considered a bad coding practice because anyone who is using such a code has to find out what all those parameters mean so that he can pass the correct values in correct sequence. And the problem is compounded if your arguments are of similar or same datatype. For example, in the product constructor here I have four different numbers that need to be passed. Now, some of you may be thinking that the parameter name themselves can be a documentation, but that is not the case. Usually we distribute our code in a compiled format, in a jar file, to other developers. So the parameter name itself is of no use. The only way they can find out about these parameters is to refer to your code documentation. Now, builder design pattern can really help us in these situations. It will help us in two ways. The only way they can find out about these parameters is to refer to your code documentation. Now, builder design pattern can really help us in these situations. It will help us in two ways. First of all, it will make it easy to use such constructors so that we can create objects of this class. And second, it will help us avoid writing such constructors in first place and still have our objects immutable.

Now let's look at a different problem.



What problem builder design pattern solves? --> Objects that need other objects or "parts" to construct them.

class Address {

public Address(String houseNumber, String street,....) {

//initialize

}

}

class User {

public User(String name, Address address, LocateDate birthdate, List<Role> roles ) {

//initialize

}

//other code

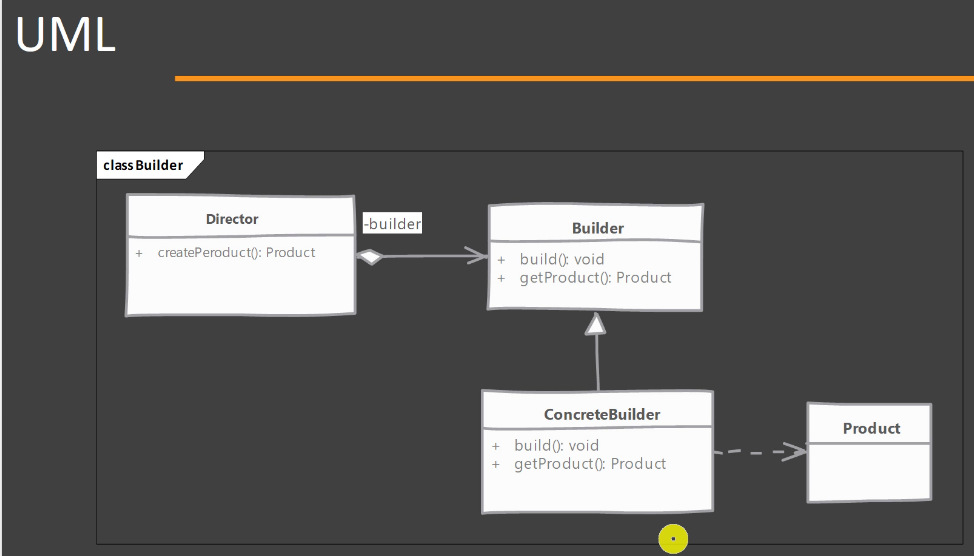
}

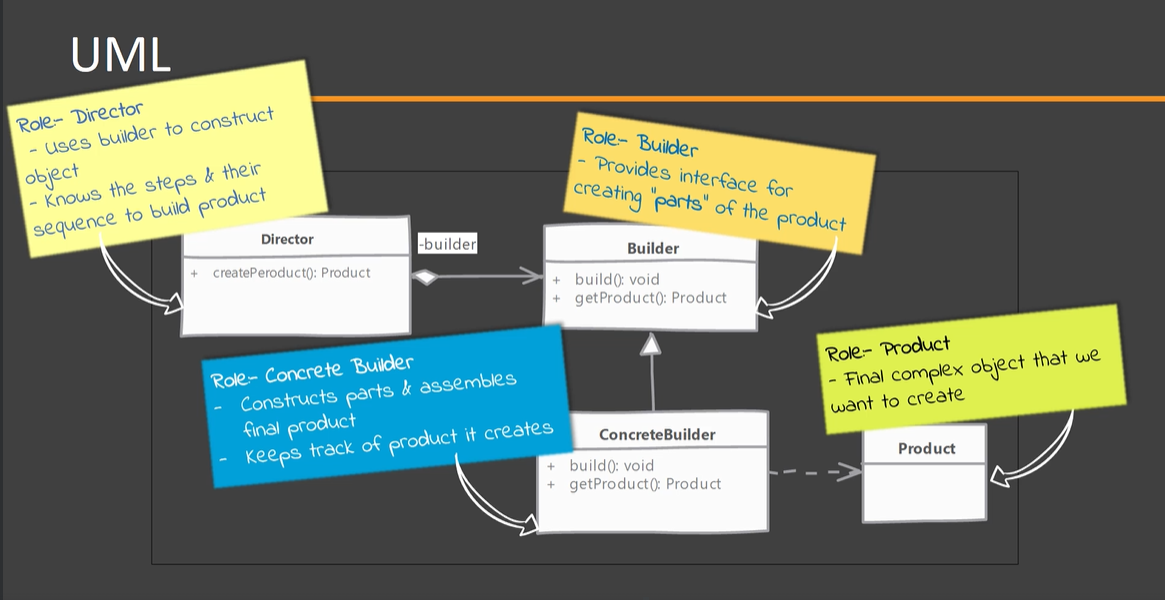
I have two classes. I have an Address class and a User class. Now if you look at the constructor of User class, you will see that I need an Address object, a collection of Role class objects in order to create my User object. So you can think of these objects, the Address object and the Role's collection as the parts that we use to assemble our User object. So whenever you have such a requirement that you need multiple parts in order to build your object, you can use builder design better. Second point is that if you look at this User constructor, you will know that we have to first create the Address object. Then we have to create multiple Role objects and then a collection which holds on to these Role objects. And then we can call our User constructor. So there are certain steps that need to be followed before we can create our User object. And here again, the builder design pattern is of great use to us.

When to use builder design pattern?

So we will use builder design pattern whenever we have a complex process to create an object. And the complexity could be because of the number of arguments that a constructor needs or the complexity may be because of the steps that are needed in order to create the object. Now, what this pattern does is that we remove this logic that is related to creating an object/instance from our code and we abstract it out into multiple separate classes so that the user/client of our object can easily create objects.

Now let's look at the UML diagram of the builder design pattern.





You will see that we have four different roles in this design pattern and we are going to look into each one in detail. So first of all, we have a Product object or Product class and we want to create objects of this class. So this is our end result. This is what we are going to build in our builder. And creating a Product object by itself is a complex process. So next, we have a Builder. Now Builder will define methods that will allow the user/client to build parts of our object. So if you look at our first example where we had multiple constructor arguments. Builder allows you to specify these arguments or these values one at a time. So that makes it easy to create our final Product object. Builder also provides a method to assemble the final Product object. So here I have a method called as build() whose job is to create or instantiate the Product class. Builder can also provide a method that users/clients can use to query already built Product object. So once you have assembled the Product object, you can use this getProduct() method to get the already built Product object out from the Builder. Next, we have a ConcreteBuilder. So, if your Builder is an interface or an abstract class, then ConcreteBuilder is simply an implementation of that Builder. It will provide all the method implementations which construct the parts of our object. It will also provide a method implementation that assembles the final Product object, and it can also optionally keep track of that Product object that it has just built so that anybody can call that getProduct() method and get the already built Product object out. Finally, we have a Director. Now we have seen that builder provides these multiple methods to build parts. Now somebody has to call these methods and there could be a sequence in which these methods need to be called. So there is a logic in which the Builder has to be used, and that logic is provided by Director class. So Director knows how to use the Builder object to create our final Product object. So this is the builder design pattern.

b) Builder implementation steps:

We are going to look at what are the steps that we should follow when we are thinking about implementing a builder in our code. So we will start by studying the object that we want to build our builder for. And the goal of this study is to find out the parts or objects that are needed by our final object. It could also be to find out the steps that are needed in order to create our final object. You can look at the constructor arguments or the object state in order to find out these parts that are needed. Once you find out these parts, we will add methods in the builder that correspond to each one of these parts. So we will provide methods that build each of these parts one at a time. Next, we have to provide a method that assembles or builds our final object. Then we should provide a way or a method in order to get that finally built object outside. Typically what we do is the build() method is the one which returns the already built object. We can also provide another way to get the object out of builder. What we can do is we can keep the reference of the object that we just built in the builder and provide a getter method that will simply return that already built object. Next, we have a director. Now you will typically find that director is rarely implemented as a separate class. Clients themselves will play the role of director. That means clients will know how to use the builder that we have provided. All right. So these are the steps that we should follow when implementing a builder.

c) Builder - Example UML:

So before we can start coding in Eclipse, let's first look at what we are going to build. So here is an UML diagram of the example that we are going to try our hands on.

A diagram of a software

Description automatically generated

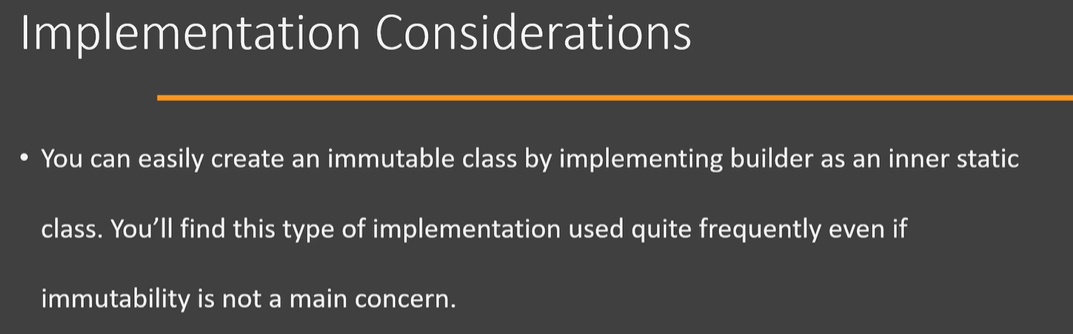
You will see that I have an interface or an abstract class called as UserDTO. So DTO stands for data transfer objects and UserDTO is a data transfer object for user entity. Now UserWebDTO is an implementation or a concrete data transfer object, and this is what we are going to build. So UserWebDTO is our final project. To create that UserWebDTO, we are going to implement UserWebDTOBuilder. So this is our concrete builder that we are going to write in our code. UserDTObuilder is an interface or an abstract builder that has simply defined the methods that correspond to each part of the UserWebDTO object. Client will be simply a main class with public static void main() method and client itself will play the role of director in our example. These classes that you see in orange colour are examples of how we can further enhance this example. So as we have provided UserWebDTOBuilder, we can also optionally provide another builder which also implements our UserDTOBuilder and build another type of object(i.e., UserRestDTO). So this is how you can extend your builder design pattern.

d) 2nd way to implement Builder design pattern:

Now, what we are going to look at is another way to implement this design pattern, which you will see commonly used in real world. So here we have a UserDTO class. Now this class is our product class. As usual, we have a product class, and we want to create objects of this class. One thing you will notice that we don't have the constructor in our UserDTO class, which needs a lot of arguments, and we are still going to create an immutable instance of this class. And to do that, you will see that we have the public getter methods that allow anyone else to read our object's state, but we have setter methods that are declared as private. So that means these setter methods are not accessible or visible outside of this class. So this way our user object is going to be an immutable object. But how does this affect the builder? Well, you will see that we are declaring our builder, which is in this case UserDTOBuilder, as an inner static class inside of our UserDTO class. And this is a common way to implement a builder. So our builder is contained within the class whose object it is going to build. So, it provides a nice namespace for our builder. That is one benefit. Second benefit is that since it is an inner static class, this builder is able to use the setter methods which are private on the UserDTO class. And in this way we can build our immutable object without writing a complex constructor or a constructor which needs a lot of information. Rest of the builder implementation remains the same. We still have methods that build parts of our final object or final UserDTO, we still have a build() method. What is changed now is that instead of using the constructor of our UserDTO class to set its properties, we are going to use the setter methods on our final object. And once we have set all the properties of our object, we are going to return that instance to the outside world. One additional change that you will see often implemented, although it is not required, is that we will have a static method inside of our UserDTO class. So inside UserDTO, we have a static method called as getBuilder() and this method simply returns an instance of our builder. Of course, since our builder is a public static class, we don't need or to have this getBuilder() method here. We can also create the builder instance outside, but it is something of a common practice that you will see followed in a lot of real world code. But if our builder class would be private static then we would definitely need static getBuilder() method inside UserDTO class.

e) Builder – Implementation and design consideration:

We are going to look at some of the points that you should keep in mind while implementing or designing builder design pattern. So to start off, let's look at some of the implementation considerations. You can easily create an immutable class if you implement your builder as an inner static class, because that way you can have your setter methods declared as private and avoid writing a bad constructor which needs all of the object state in the constructor itself. You will find that this type of implementation is always preferred way to implement the builder pattern, even if the immutability of the final object is not your main concern or which is something that you don't require. Because having a builder inside the product class gives it a nice namespace so you don't have to look through your codebase to find the correct builder. The builder is residing inside the class whose object it is going to build, so it provides a nice namespace and it is one of the preferred way to implement this design pattern.



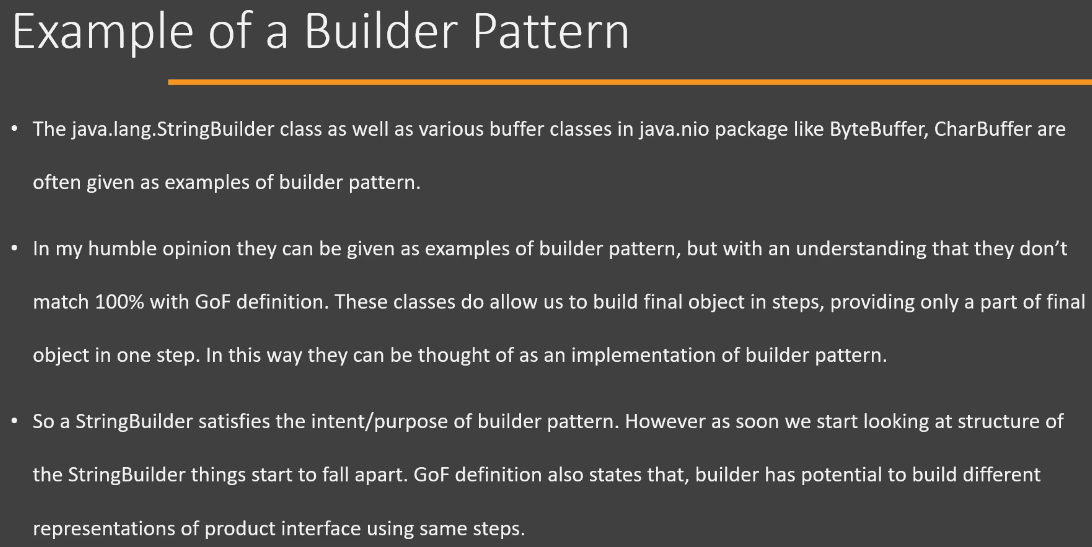
Now let's look at some of the design considerations while implementing or while you are considering builder design pattern. First of all, you will see that director role is rarely implemented as a separate class because what you will find in the real world is that the number of steps that are required to create the object and their sequence is always the same. And the code, which is going to use the final object is the one which is going to play the role of director. Abstract builder is also something that you will not see implemented because if you have a single product class without any inheritance hierarchy, then it doesn't make sense to have an abstract builder because there is going to be only one builder for our single product class. However, if you have a inheritance hierarchy for your product class, like in our example(1st implementation), we had a UserDTO interface and we had multiple data transfer objects like UserWebDTO and UserRESTDTO. Now all these data transfer objects relied on the UserDTO to build themselves. So there it made sense that we have a abstract builder and then multiple concrete builders(Search **builder design pattern with inheritance** in Java 🡪 <https://www.baeldung.com/java-builder-pattern-inheritance>). But always remember that abstract builder is optional and you should choose whether you want an abstract builder or not based upon the hierarchy of the class whose object you are going to build. And if you're running into a too many constructor arguments type of a problem, for example, you are designing your classes and you have a class whose constructor needs a lot of information, then that is a good indication that a builder design pattern can help. Of course, that's just an indication and the problem may lie somewhere else as well.

A screenshot of a computer

Description automatically generated

f) Builder - Real world examples:

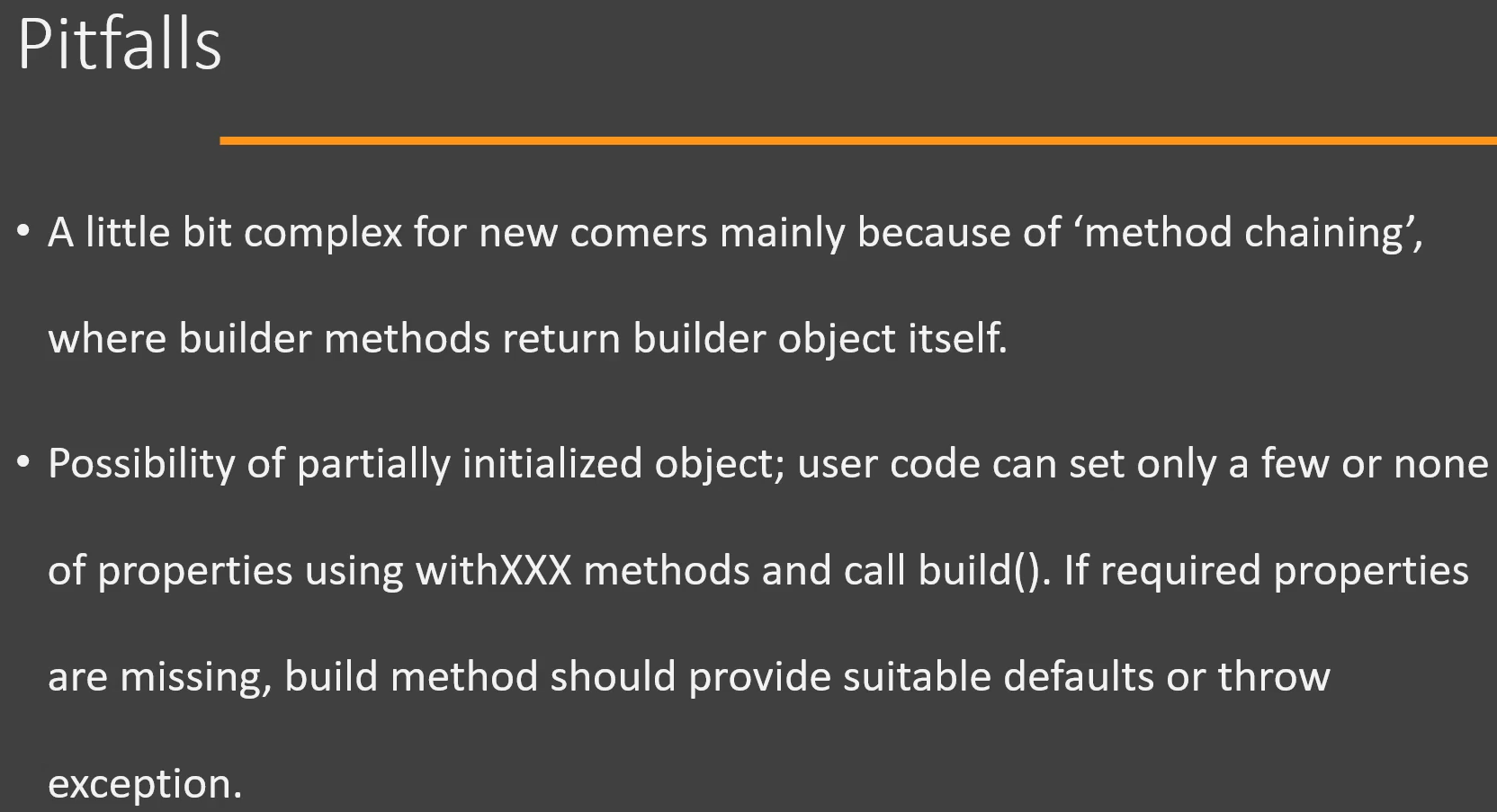
It's time to look at some real-world applications of builder design pattern. Now, what we are going to do is we are going to look at the classes that are shipped with Java development kit or classes which are present in these popular frameworks like Spring, Struts and Hibernate, and we are going to try to find application of builder design pattern. So you might have heard that StringBuilder and various buffer classes in java.nio package are examples of builder pattern and in my opinion they can be given as an example of builder pattern with an understanding that these classes do not match 100% with the definition of builder pattern given by ‘Gang of Four’ book. Because these classes do allow us to build our final product/object in steps. So we have the append() method in StringBuilder and in each invocation of that append() method, we provide only part of the final string. So in this way these classes can be thought of as an implementation of builder pattern. So StringBuilder satisfies the intent or purpose of builder pattern. However, as soon as we start to look a little bit deeper into the implementation and capabilities of these classes, then this particular definition or example starts to fall apart. Because one of the requirements that is given by the definition in the ‘Gang of Four’ book for builder pattern is that builder has potential to build different representations of product interface using same steps. So if you remember the UML diagram that we studied, we saw that our final product/object requires objects of different classes and those can be built by our builder using the same steps. Suppose our final product/object requires an instance of UserDTO(which is an interface) type. Now, client can pass any implementation of UserDTO interface, builder will be able to build that part of the final object. And this is where StringBuilder is not conforming with the definition of builder pattern. However, if you're sitting in an interview and somebody asks you to give an example of builder pattern, I will still start by telling them about StringBuilder. And you can add that, an argument can be made that StringBuilder is not 100% a builder implementation or builder pattern.



So let's look at another example where we don't have this kind of a problem, which is a clean example of builder design pattern in Java API, and that is the Builder class which is provided in the java.util.Calendar class. And remember that this Builder class is provided or is available since Java 8. So below is a code from that particular class. So remember this code is taken from the Builder inner static class in the Calendar. So you will see that we have a Builder class and this class has provided methods that allow us to build parts of our final Calendar instance. So it has methods which allow us to set week-date or set individual fields of the calendar, so we can set year in one step, we can set date in one step, so on and so forth. And it has provided the build() method, which is our assembly method. And once we call build(), the Builder is going to use the parts that it has built and give us an instance of Calendar. Okay, so Calendar.Builder is a great example of builder design pattern in Java API.

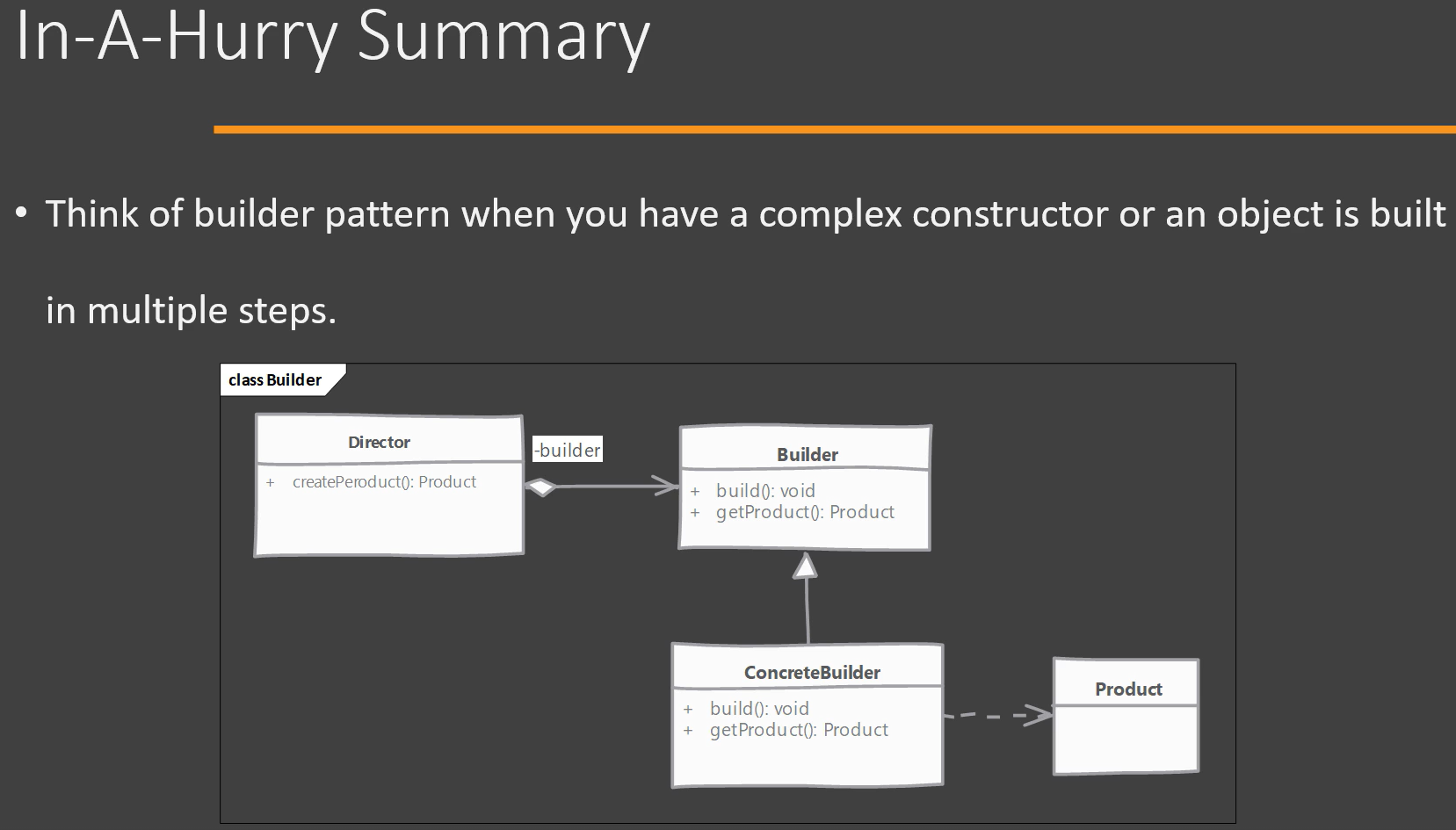
g) Builder – Pitfalls:

We're going to discuss the pitfalls of builder design pattern. So if this pattern is implemented correctly and used correctly, then there are no pitfalls to this particular pattern. So let me explain on this particular bit. So when we are discussing pitfalls of a particular design pattern, we are considering things like number of new classes that had to be added, the impact or the effort that is required to refactor your existing code, the complexity to understand a particular pattern. If we consider those aspects, then this pattern builder design pattern does not have any pitfalls. But this does not mean that this pattern is completely foolproof. There are some issues that you can face when working with this pattern. So let's discuss those issues here. The first issue is specifically to those who are completely new to programming or completely new to Java, and they may find it a little bit difficult to understand about method chaining. So method chaining is quite frequently used with this particular pattern where we chain one method after another. Okay, so this can be a little bit of an issue if you are a newcomer. The next issue that you can face is that there is a chance that you can run in a scenario where you have a partially initialized object. So in this scenario, what can happen is that the client code or the user code, which is using our builder, can set only a few or none of the properties on our builder, and it can directly call the build() method on it. And if the required properties are missing, then build() method needs to explicitly handle that particular scenario. So you can either provide some sensible defaults or you can throw an exception. So this is some additional handling that is required if there is a chance that user code can directly call your build() method. Okay. So these are few issues that you can face when working with builder pattern.



i) Builder – Summary:

We are going to look at a summary of builder design pattern. So you will use builder design pattern whenever you have a class with complex constructor or creating an object requires multiple steps like creating several other objects that are needed by our final object. In those situations, builder design pattern can help us by removing the complexity of creating an object from our code into a separate class. Now below is the UML diagram of the builder design pattern. We have four different roles in this design pattern. First, we have a Product class and we want to create objects of this class in our builder. Then we have a Builder which defines methods to build parts of our final object, one part at a time. Builder also provides a method to assemble or instantiate the final object using the parts that we have already built. Builder can also provide an optional method that anybody can use to query or get an already built object from the Builder. If your builder is an interface or an abstract class, then you will also have a ConcreteBuilder. Now this class is simply going to implement all the methods that are defined in the Builder. Lastly, we have a Director. Now, Director knows how to use our Builder class. It knows about the steps that are needed to create the final object, and it also knows about the sequence in which these steps need to be executed. Now, one thing to note here is that you will rarely find Director implemented as a separate class. Typically, the class who is going to use our final product is going to play the role of Director.



Now let's look at the two implementations of builder design pattern that we have seen. So here I have a UserWebDTOBuilder, which is our builder class, and you will see that we provide methods like withFirstName(), withLastName(), which build part of our final object. A common way to implement builder is that these methods which build part return the builder instance itself. This allows the user to chain these methods one after another. Next we have a build() method, which is the assembly method that is going to take all the parts that we have built so far and instantiate the final product object. We also have this optional method implemented here called as getUserDTO() and this method is going to simply return an already built object from this builder. Then we have a client. Now, in order to use this builder, we are going to need a Director. So director is typically implemented as a method in our client. Director will need access to the builder instance and any information that is required to build the final object, we will call those methods which build parts of our final object on the builder instance and finally assemble the object and return it. Client will be in charge of instantiating or creating instance of the builder and it will pass this builder instance along with any information needed to the director in order to get the final product outside.

A screenshot of a computer

Description automatically generated

Now here is another way to implement Builder and that is implementing Builder as an inner static class. So here we have a product class i.e., UserDTO, that means we are going to create objects of this class in our builder. To create this class as an immutable class, what we do is instead of writing a complex constructor that needs a lot of arguments, we provide setter methods for setting the states or properties of this userDTO class. But we mark these setter methods as private, making this class effectively immutable. Now, since our builder is an inner static class, it can still use these setter methods in order to set the properties of this class. So here we have builder as an inner static class and our product class i.e., UserDTO can also provide a static method that returns instance of this class i.e., builder class. So this is a common way to implement builder that you will find in use. In these situations, the client remains the same. The only change that you will see is that now we are going to use the static method on our product object to get the builder instance. The director, which is using this builder, will work in the same way.

A screenshot of a computer

Description automatically generated