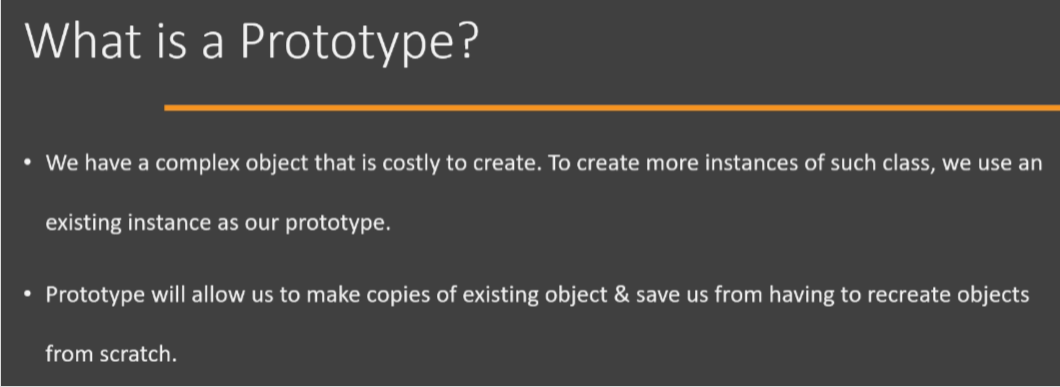
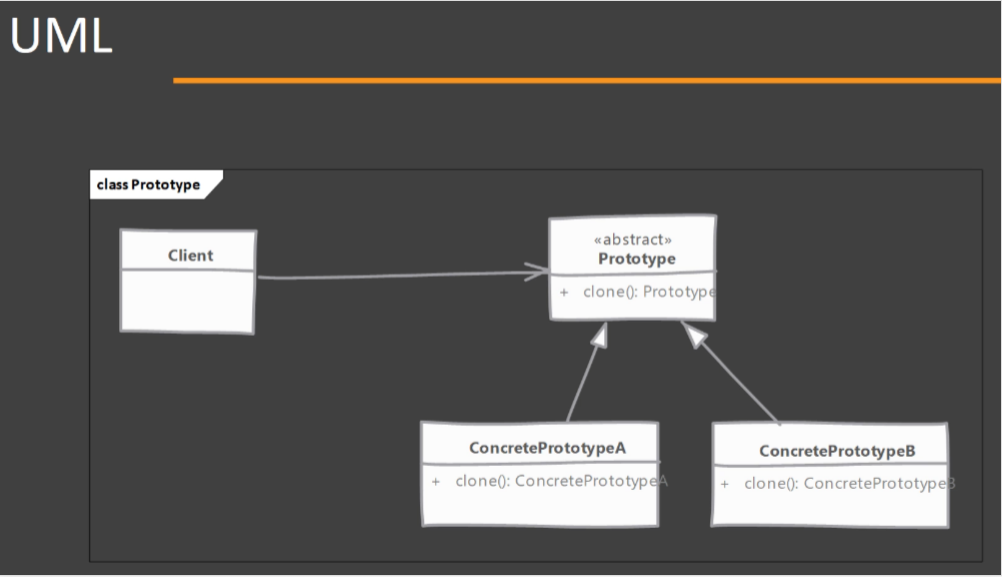
Prototype Design Pattern

1. Prototype – Introduction:

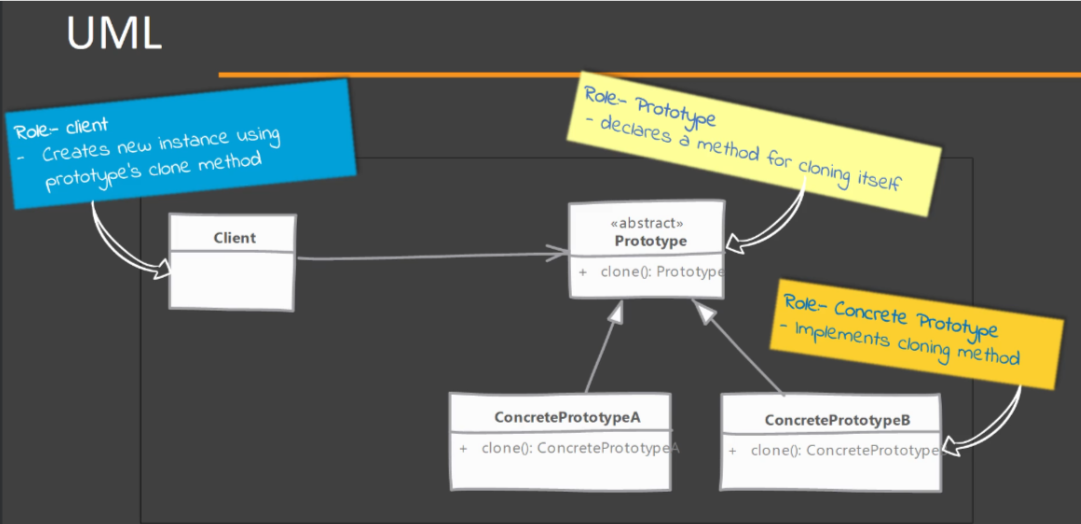
Now we are going to learn about a creational design pattern known as prototype. So, what is this design pattern and why we use it? So, whenever we are in a situation where we have an object and creating an object is a costly operation and the cost may be the performance cost(may be some calculation is needed to create that object), or the cost may be related to using external resources which are not in our control. So, whenever we have such an object, then to create multiple instances, we use that single object or existing object as our prototype. And this pattern allows us to make copies of that existing instance and it saves us from creating these instances from scratch. This is a very simple design pattern, especially in Java, and we are going to see how that is so.



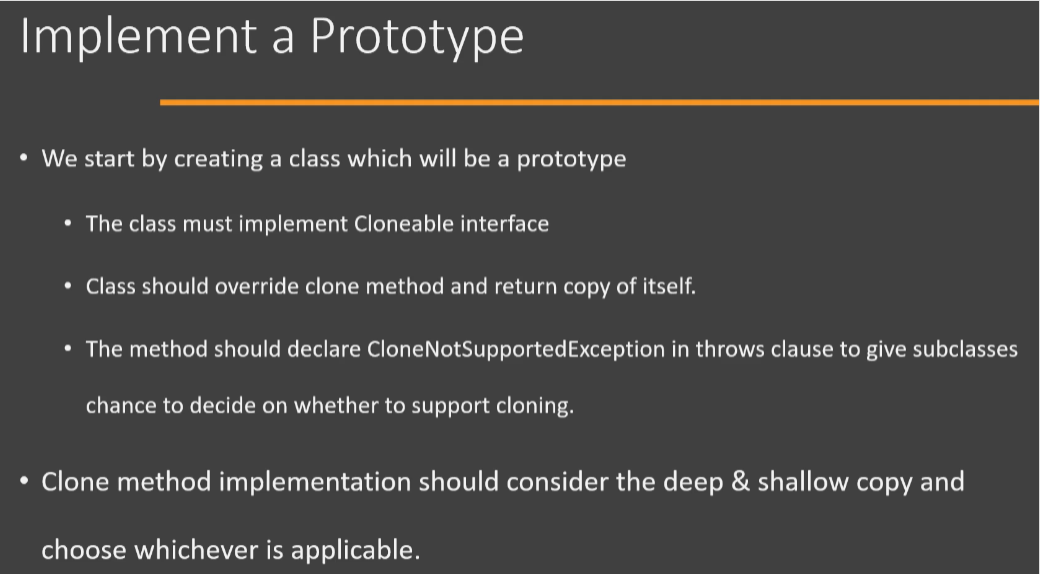
So here is the UML of prototype design pattern. It's quite a simple UML diagram. First of all, we have a class that will be the prototype(i.e., Prototype class), so objects of this class(i.e., Prototype class) will serve as a prototype for our use. So we are going to get hold of one object of this class(i.e., Prototype class). And from that single object we are going to create multiple copies.



Now, this class(i.e., Prototype class), defines an operation which allows us to create copies in Java. That operation is a clone operation. So if you remember, Object class in Java has already defined a method, a protected method called as clone(). Now, purpose of that method is to make copy of an existing object. Now, once we have a prototype and let's say if it is an abstract class, then we will have multiple concrete classes that implement that clone() method. And this clone method make copies of the object and the client will use these classes(i.e., ConcretePrototypeA and ConcretePrototypeB) or object of these classes and create multiple instances. So this is the UML diagram. It's quite a simple design pattern, especially in Java.



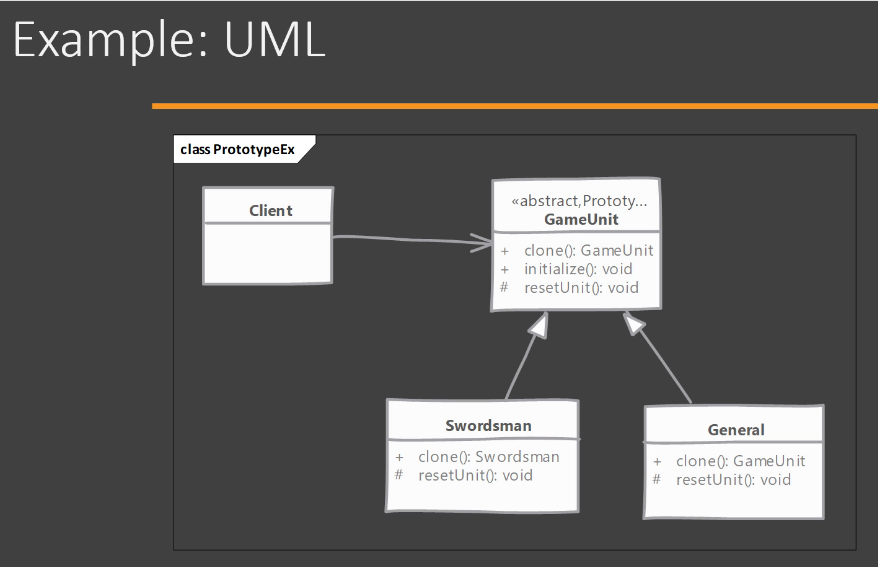
1. Prototype - Implementation steps:



Now let's look at some of the steps that we need to follow in order to implement prototype design pattern. Now, implementing prototype in Java is quite easy. We start by creating our class, and objects of this class are going to be our prototype objects. Now this class must implement the Cloneable interface. So Cloneable is an inbuilt Java interface and it is a marker interface. A marker interface is an interface which doesn't define any methods but serves as a marker or indication of capability of our class. So having implemented Cloneable, you are telling other code that your class supports the clone operation. Next step is obviously to override the actual clone() method. Now this method is defined in the Object class and it is a protected method in Object class, so you may want to make that a public method when you are overriding that method. So that the code which is in a different package, can call the clone() method on your objects. Now the method signature in the Object class has a ‘throws’ clause in the clone() method and it declares CloneNotSupportedException in its ‘throws’ clause. And you may want to keep this ‘throws’ clause because in future you may run into a situation where you have a subclass that doesn't support cloning. Having this ‘throws’ clause will tell the calling code that there may be a chance that the operation may fail. Now, while implementing the actual clone() method, you should take into consideration whether you need a deep copy or a shallow copy. In shallow copy, what we do is we simply copy the object's properties into the new object. In deep copy, we will create all the objects that are needed by our prototype object.

1. Prototype – Hands-on Example UML:

Let's go over the UML diagram of the example that we are going to implement in Java Code. So we're going to start with GameUnit class, this will be the prototype class and objects of this class will support the cloning operation. So we will have a clone() method and you will see that we also have this public initialize() method and we're going to see why we need this method whenever we are implementing prototype design pattern. There will be two subclasses or child classes of our GameUnit. One is the Swordsman and another class will be General. Now what we are going to do is we're going to have Swordsman which will support the cloning and we will have the General subclass which will not support cloning operation. Then we will have a Client which is going to create the initial object and then clone from that existing object in order to use our prototype design pattern.



1. Prototype – Implementation:

Refer to prototype-end-demoed folder under Creational\_design\_pattern/Prototype\_design\_pattern for code.

Now we're going to see how we can implement prototype design pattern. And as usual, I have written some code that we are going to use while implementing this design pattern. So we are going to go over the classes which I have already written. These classes will correspond to the example UML that we just saw. So first of all, we have a GameUnit class and this is a base class for all of our prototype classes. Now, GameUnit, the object of this class represents some unit of a game which is played on a map. This class has a single property or attribute called as 'position', and this represents the position of this particular GameUnit object on a map. Then we have two child classes of this GameUnit. First of all, we have a Swordsman class which is extending from GameUnit. It defines its own property called as 'state', which refers to the current state of this particular Swordsman. And we have another class called as General, which is again extending from GameUnit. What we are going to do here is we're going to say that General class object don't support cloning. Maybe the General unit is a unique unit in our game and we do not want to accidentally create copies of this particular object. So let's start implementing our prototype design pattern. To begin, we are going to implement a marker interface called as Cloneable. So Cloneable is an inbuilt Java interface, which indicates to the user of this class that this particular class supports the cloning operation. Invoking Object's clone() method on an instance that does not implement the Cloneable interface results in the exception i.e., CloneNotSupportedException being thrown. By convention, classes that implement this interface i.e. Cloneable, should override Object.clone (which is protected) with a public method. See Object.clone() for details on overriding this method.

@Override

protected Object clone() throws CloneNotSupportedException {

return super.clone();

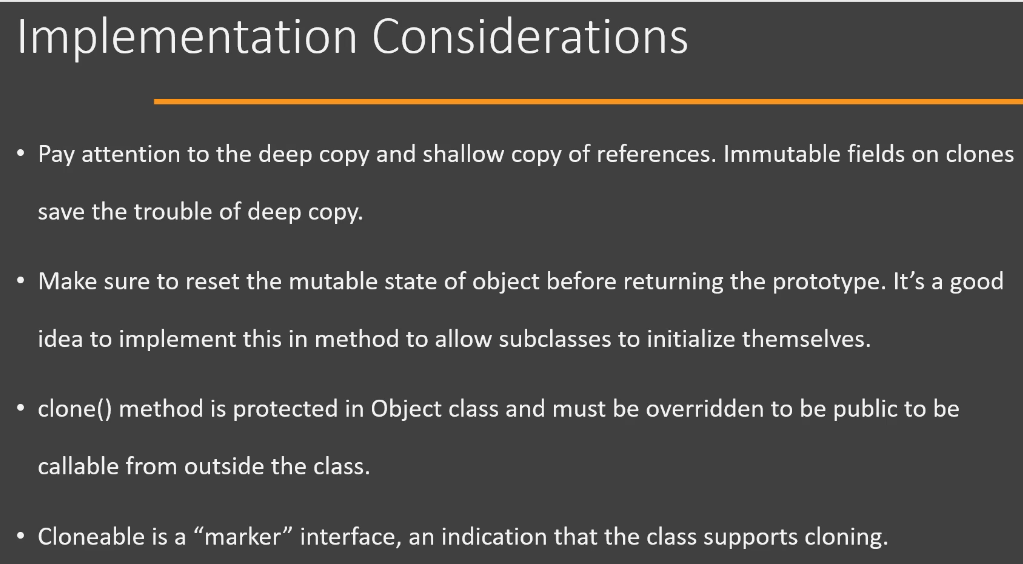
}

Now the actual method which clones our object, we are going to override the clone() method from Object class and this method is going to provide the copy of this particular class(i.e., GameUnit type class) object. Now we're going to do two changes to this method. First of all, we are going to make this method public so that it can be used by classes which are outside of this particular package(in our case, package is com.coffeepoweredcrew.prototype). Then we are going to use the covariant return type, which is a feature of Java, where we can return a subclass or child class of Object. And here we're going to return GameUnit. You will see we have a 'throws' clause here and we are saying that CloneNotSupportedException is thrown from this method. We are going to keep this 'throws' clause so that we can use that in classes which do not support cloning. Now let's start the cloning operation. The first thing that you need to consider is whether you need a shallow copy or deep copy of your state. If you look at the state of our GameUnit, we'll see that we have only one property in our class called as 'position' which is a type of Point3D. Now, Point3D class, which is from javafx.geometry package, is an immutable class. That means once you create an object of Point3D class, you cannot modify it, which means we can get away from shallow copy. The code which we have inside our clone() method, inside GameUnit class, our Object's clone() method will be doing shallow copy always. Since we can't change an existing object of Point3D class because it is an immutable class(either it has no setter methods or has private setter methods, it has private final field members, may be it is a final class), we have to create a new object of Point3D class and assign it to the 'position' attribute of the cloned object, which we are doing inside the initialize() method. This way we are making the deep copy of our GameUnit class type object. Hence, ultimately our clone() method, inside GameUnit class, is making the deep copy of the object of type GameUnit. Now to do a shallow copy, we are going to use the clone() method from the Object class. So we are going to simply say 'super.clone()' and this will call the clone() method implementation that is available in Object class. We need to store the returned object to the reference variable(I have named the variable as unit). Now the clone() method returns an Object, so we need to cast the return value to GameUnit. So now we have another object of type GameUnit stored in 'unit' variable, which is a copy of 'this' object and it is a shallow copy. Next, even though the 'position' attribute/field of cloned object(i.e., GameUnit class type object) is a shallow copy, we need to make sure that our cloned object state is reset when we are returning from our clone() method. So 'this' object(i.e., GameUnit class type object) inside our clone() method may be located to some location on the map. And when somebody clones 'this' object(current object), we want to make that copy located at the origin. So in order to reset the state of our cloned object, I'm going to write another method here, a protected method called as initialize(). So initialize() method will have one job and that is to reset the state of the current object(i.e., cloned object). So it's[initialize()] going to say 'this.position=' and in order to get Point3D object which is located at origin, we can use the constant which is defined in the Point3D class called as ZERO. So this way we have reset our 'position' attribute to origin. Only thing is, we need to call this method[initialize()] on the cloned object inside our clone() method. Now, we have reset our cloned object and I'm going to return that to the outside world[from our clone()]. Now, you may think that this is enough for making our individual child classes(i.e., For example, Swordsman class) as a prototype but if you look at our child classes(i.e, suppose Swordsman), you will see that they have their own state. So we have a property called 'state' here and they(child classes) need a way to reset that property because, at the end, we will be cloning a Swordsman object only. And the implementation of clone() method which is provided in GameUnit class is resetting only the properties or state which is defined in the base class(i.e., GameUnit class). So one way is that they(child classes) can override this initialize() method of GameUnit class, but that way they need to now handle the resetting of state that is defined in base class(i.e., GameUnit class). Instead of that, what we're going to do is we're going to define one more method called as reset() inside GameUnit class. And instead of providing an implementation, we're going to declare it as an abstract method. So this way the child class has to provide an implementation for reset() method, and we're going to call reset() method from our initialize() method inside GameUnit class. In short, we used the clone() method from the Object class because we wanted a shallow copy. We wrote a new method called as initialize() which would reset the state that is defined in this base class. We have defined an abstract reset() method that subclasses or child classes will override. And we're going to call that reset() method from our initialize() method. All right, so this is how you can implement our clone().

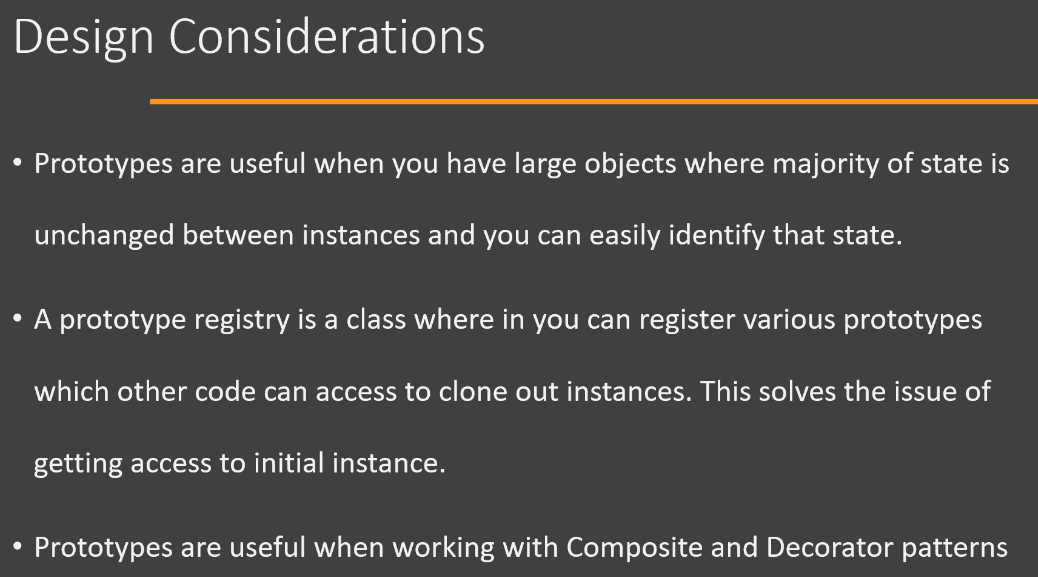
Now let's go to the Swordsman class and here we need to now override the reset() method from the GameUnit class. Inside reset() method, the only thing that we need to do is to make sure that the state that is present in our Swordsman class is reset. So I'm going to simply say state="idle". So even if somebody calls the attack() method on our Swordsman object, it's state will change to 'attacking'. And if we call clone() on the same Swordsman object then it's going to create a new Swordsman object which is in idle position. Same thing can be done into General class, but we discussed that we do not want to support the cloning operation of General object. So how can we do that? Well, we are going to override the clone() method from our GameUnit class. We also need to override the reset() method because that is an abstract method inside GameUnit class. But in the clone() method of General class, we are going to throw 'new CloneNotSupportedException()' and we can pass along a message saying 'Generals are unique'. In the reset() method of General class, we are going to throw 'new UnsupportedOperationException()'. This reset() method should not get called on General object, but still we are going to throw an exception to indicate that reset is not supported operation on General object. So in this way we can have a class that extends from our prototype base class but still decide that it doesn't want to get cloned. In that case, it will override the clone() method from our base class and throw the CloneNotSupportedException. And this is why we have this 'throws' clause because the client, which is going to use these classes(i.e., Swordsman and General), is aware that there could be a subclass which can decide to throw this exception. Now the only thing remaining is to go into our Client code. We have a main() method here and we are going to first create the initial instance of our Swordsman class by using the constructor. Then we are going to change some of the properties of this instance. So we have now moved our Swordsman initial object and let's call the attack() method here, and then we are going to print the current state of our Swordsman initial object. Now let's create another Swordsman object s2. But here we are going to use our existing instance s1 and call the clone() method. Now, remember, we need to cast the return value of our clone operation to a Swordsman. And we also need to handle the CloneNotSupportedException. So for time being, I'm going to add the 'throws' clause in my main() method so that we have a much cleaner code and let's check the state of our cloned object by running the program/code. So this is how you can implement prototype design pattern in your code.

1. Prototype – Implementation and Design considerations:

Let's look at some of the implementation and design considerations when working with prototype design pattern. So first point is that you should always pay attention to whether you need deep copy or shallow copy in your clone method. Now, if your prototype class has objects as its state/field or they're part of your prototype state and they're immutable, then you can get away with shallow copy because these objects are immutable. So a copy can keep pointing to those same objects. However, if you have mutable objects as part of your prototype state/field, then you may need a deep copy. So always pay good amount of attention to these deep and shallow copy options. Next, make sure that when you are returning the copy from your clone method, you are resetting the mutable state of your object. Otherwise, the code that relies on these objects may behave differently. Now, it is always a good idea to implement this reset operation as a separate method so that subclasses can override this method and provide their own code to reset their own state. Now the clone method is protected in Object class, so always remember that. So when you're overriding, make sure that you are making it public so that it is usable from different package. Now Cloneable itself is a marker interface, so it doesn't define any methods, but it indicates that your class supports the clone operation.



Now let's look at design considerations when working with Prototype. Prototype is really useful if you have a large object, that means the object has multiple fields in it as part of its state, and majority of that state remains unchanged between instances. And you can easily identify whether which part of your state does not need change and which part is going to change for multiple cloned instances. Now, the main another point that you should consider is how to get hold of that initial instance so that you can make copies of that object from anywhere in your code. To do that, you can also implement a prototype registry class where you can register the instances of your prototype and other code can access this central location to get your initial copy and then clone that to create other instances. Now, this implementation solves the problem of getting hold of that initial instance. Now, prototypes are really useful when we are working with composite and decorator design patterns that we're going to see in future lectures.



1. Prototype – Real-world example:

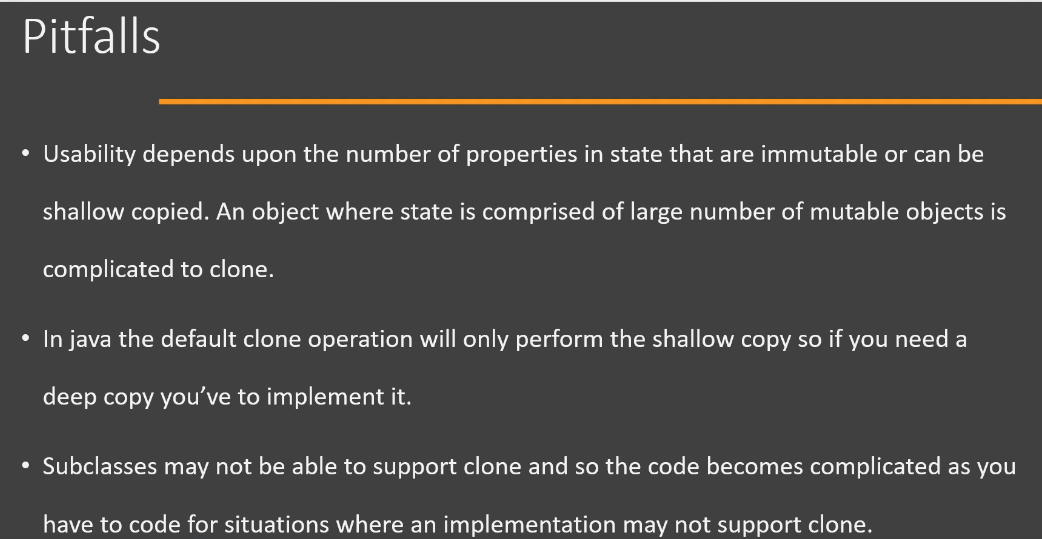
Let's try to look at some real life examples of prototype design pattern. Now, unfortunately, there aren't many places in the Java class library or other popular frameworks where prototype design pattern can be found to be used. So the only example that we can study or that we can look at is the actual clone() method that is present in the Object class of Java class library. Because if you think about it, this method is actually provided to us by the Java and this method has capability to clone an existing object, thus allowing any object to act as a prototype. Of course, before we can actually do the cloning operation, we need to make sure that our class is implementing the Cloneable interface and we have the access to the clone() method that is defined as protected in the Object class. However, this method does work as a prototype. That means it makes a copy of an existing object that we can use instead of calling the constructor of that particular class. All right, so Object.clone() can be given as an example of prototype design pattern.

A screenshot of a computer

Description automatically generated

1. Prototype – Pitfalls:

Let's look at some of the pitfalls of using prototype design pattern. Now, feasibility of using prototype design pattern depends upon the prototype state. If the state of your prototype object is made up of large number of mutable objects that needs change, then implementing prototype design pattern can be a complicated process because now you have to provide deep copy of all those mutable objects which are present in your prototype object state. The default clone() method implementation, which is present in Object class, it only performs the shallow copy. So whenever you need a deep copy, you have to provide implementation for that deep copy yourself. Next, subclasses of your prototype can decide that they will not support the cloning operation or not. And in such a case, they will throw a CloneNotSupportedException. So the code, which is using your prototype design pattern has to handle such subclasses where cloning is not possible and they have to code an alternate way to get hold of your subclass instances. So the code, which is using your prototype design pattern now becomes complicated. So these are some of the pitfalls of prototype design pattern.



1. Prototype – Summary:

Let's look at summary of prototype design pattern. You will use prototype design pattern whenever creating a new instance of a class is a costly operation, or in some cases it may not be possible because the object is directly supplied to your code. In Java, we implement this pattern using the clone() method, which is defined in the Object class. So we override that method in our class and provide the implementation to return copy of our existing object. Now objects which have majority of their state as immutable(doesn't require change) are good candidates for prototype. Having a large mutable(requires change) state makes the cloning operation that much costly and difficult to implement. Now, when implementing clone method, you have to pay attention whether you need a deep copy or a shallow copy of your prototypes state. Also, we have to ensure that the cloned object that we are returning from our clone method is reset or initialized with appropriate state values so that the code that is using our cloned copy doesn't get affected by leftover values for state.

