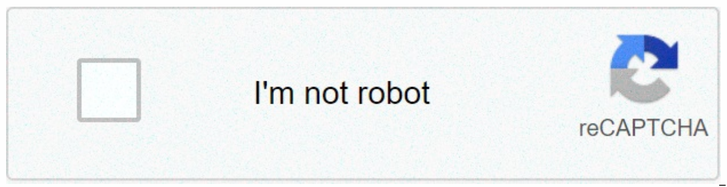

Continue

Adapter pattern example in android

The adapter design pattern is one of the structural design pattern and is used so that two unrelated interfaces can work together. Object adapter that combines this unrelated interface. Adapter Design Pattern Adapter design pattern is called one of the first in great real life mobile charging. The mobile battery needs 3 volts to charge but the regular socket produces either 120V (US) or 240V (India). Thus, the mobile charger works as an adapter between the mobile charging socket and the wall socket. In this tutorial, we will try to implement the multi-adapter using the adapter design pattern. So first of all we will have two classes - Volt (to measure volts) and Socket (producing 120V constant volts). package com.journaldev.design.adapter; public class Volt { private int volts; public Volt(int v){ this.volts=v; } public int getVolts() { return volts; } public void setVolts(int volts) { this.volts = volts; } } package com.journaldev.design.adapter; public class Socket { public Volt getVolt(){ return new Volt(120); } } Now we want to build an adapter capable of producing 3 volts, 12 volts and the default 120 volts. Therefore, first of all, it will create an adapter interface with these methods. package com.journaldev.design.adapter; Common interface SocketAdapter { public Volt get120Volt(); general Volt get12Volt(); general Volt get3Volt(); } When applying a Bidirectional Adapter Pattern Adapter pattern, there are two approaches – class adapter and object adapter – but both of these approaches produce the same result. Class Adapter - This form uses java heredity and extends the source interface, in our case in the Socket class. Object Adapter - This form is used in Java Composition and contains the adapter source object. Adapter Design Pattern – Class Adapter Is where our adapter implements a class adapter approach. package com.journaldev.design.adapter; Heredity-using socket adapter for adapter pattern{ @Override general Volt get120Volt() { return getVolt(); } @Override public Volt get12Volt() { Volt v= getVolt(); return convertVolt(v ,10); } @Override public Volt get3Volt() { Volt v= getVolt(); returnVolt convert(v,40); } private Volt convert(v,40); } private Volt convert(v,volt, Volt, Volt, volt, int i) { return new Volt(v.getVolts()/i); } } Adapter Design Pattern – Object Adapter ApplicationThing is the Object adapter application of our adapter here. package com.journaldev.design.adapter; public class SocketObjectAdapterImpl applies SocketAdapter{ //Adapter pattern using composition for custom Socket socks = new Socket(); @Override general Volt get120Volt() { return sock.getVolt(); } @Override public Volt get12Volt () { Volt v= sock.getVolt(); return convertVolt(v,10); } @Override public Volt get3Volt() { Volt v= sock.getVolt(); return convertVolt(v,40); } private Volt convertVolt(volt, int i) { return new Volt(v.getVolts()/i); } } Note that both adapter applications are almost the same and that they implement the SocketAdapter interface. Adapter interface, to be an abstract class. Here is a test program to consume our adapter design pattern application. package com.journaldev.design.test; import com.journaldev.design.adapter.SocketAdapter; import com.journaldev.design.adapter.SocketClassAdapterImpl; import com.journaldev.design.adapter.SocketObjectAdapterImpl; import com.journaldev.design.adapter.Volt; public class AdapterPatternTest { public static void main(String[] args) { testClassAdapter(); testObjectAdapter(); } private static void testObjectAdapter() { SocketAdapter sockAdapter = new SocketObjectAdapterImpl(); Volt v3 = getVolt(sockAdapter,3); Volt v12 = getVolt(sockAdapter,12); Volt v120 = getVolt(sockAdapter,120); System.out.println(Object Adapter using V3 volt=+v3.getVolts()); System.out.println(Object Adapter using V12 volt=+v12.getVolts()); System.out.println(Object Adapter using V120 volt=+v120.getVolts()); } custom static void testClassAdapter() { SocketAdapter sockAdapter = new SocketClassAdapterImpl(); Volt v3 = getVolt(sockAdapter,3); Volt v12 = getVolt(sockAdapter,12); Volt v120 = getVolt(sockAdapter,120); System.out.println(v3 volt=+v3.getVolts()); System.out.println(v12 volt=+v12.getVolts()); System.out.println(v120 volt=+v120.getVolts()); } custom static Volt getVolt(SocketAdapter sockAdapter, int i) { switch (i){ case 3: return sockAdapter.get3Volt(); case 12: return sockAdapter.get12Volt(); case 120: return sockAdapter.get120Volt(); default: return sAdaockpter.get120Volt(); } } } } When we run above the test program, we follow the output. v3 volt using Class Adapter=3 v12 volt Class Adapter using Class Adapter=Class Adapter using 120 v120 volt=Using 120 v3 volt Object Adapter=3 v12 volt Object Adapter=12 v120 volt Object Adapter=120 Adapter Design Pattern Sample JDK Some adapter design pattern examples can easily be found in JDK classes;java.util.Arrays#asList()java.io.InputStreamReader(InputStream) (a Reader)java.io.OutputStreamWriter(OutputStream) (returns a writer)All for this java adapter design pattern. Design patterns provide a reliable and easy way to follow proven design principles and write well-structured and protectable code. One of the popular and frequently used patterns in object-oriented software development is the adapter pattern. Follows Robert C. Martin's Dependency Inversion Policy and allows you to re-use an existing class so that it does not implement an expected interface. If you do some research on the adapter pattern, you find two different versions of it: the class adapter pattern that implements the adapter using heredity. An object adapter pattern that uses composition to refer to an instance of the wrapped class within the adapter. We are probably aware of all the discussions about inheritance vs composition. The composition provides more flexibility and, where necessary, the unexpected side existing code. Therefore, the object adapter pattern is a much more popular approach and I am one to dread in this article. Adapter PatternA general idea of an adapter in software development is the same as in the physical world. If you've been to different countries, probably a lot of them recognized if you're using different shaped power sockets. Quite often, they are shaped in a way that does not fit the plug of the electrical device. So, how do you connect the charger of your mobile phone or laptop to this power outlet? The answer is simple. You get an adapter that you can put in the power outlet, and then you put your plug on the other end of the adapter. The adapter changes the shape of your plug so you can use it with a power socket. In this example and in other cases, the adapter does not provide additional functionality. It lets you connect your plug to the power outlet. Adapter Pattern applies the same idea to object-oriented programming by introducing an additional adapter class between the interface and an existing class. The adapter class implements the expected interface and references an object of the class you want to re-use. Methods defined by the interface call one or more methods on the referenced object and return the value of the expected type. By doing so, it allows you to re-use existing, incompatible applications by implementing an adapter class interface, fulfilling the expected contract. Let's apply the pattern to an example. I want to start the morning with a fresh cup of coffee in the coffee indering using adapter PatternI. The only problem is, I have to get out of bed and get some coffee before I drink. It would have been better if it had been prepared automatically when my alarm went off. Let's create a small application for this. An app for infapping a coffeeFilterCoffeeApp does exactly that. As a constructor parameter, the FilterCoffeeMachine interface uses theCoffee method to prepare an application and prepare the filter coffee to infuse a cup. public class FilterCoffeeApp { private Logger log = Logger.getLogger(FilterCoffeeApp.class.getSimpleName()); private FilterCoffeeMachine coffeeMachine; public FilterCoffeeApp(FilterCoffeeMachine coffeeMachine) { this.coffeeMachine = coffeeMachine; } public Coffee prepareCoffee() { Coffee coffee = this.coffeeMachine.brewCoffee(); log.info(Coffee ready!; log.info(-> + coffee); return coffee; } The FilterCoffeeMachine interface is relatively simple. It defines the brewCoffee method that you can only call to make a cup of coffee.public interface FilterCoffeeMachine { Coffee brewCoffee(); } With this application it seems to be a good approach that allows you to use different coffee machines. The only requirement is that all classes representing a coffee machine implement the FilterCoffeeMachine interface. A basic coffee machineBasicCoffeeMachine class and FilterCoffeeApp.public class can be used by BasicCoffeeMachine { custom Configuration config; custom Map<CoffeeSelection, groundcoffee=> groundCoffee; custom BrewingUnit brewingUnit; public BasicCoffeeMachine(Map<CoffeeSelection, groundcoffee=> coffee) { this.groundCoffee = coffee; this.brewingUnit = the new BrewingUnit(); this.config = new Configuration(30, 480); } @Override public Coffee brewCoffee() { // coffee GroundCoffee groundCoffee = this.groundCoffee.get (CoffeeSelection.FILTER_COFFEE); // brew a filter coffee return this.brewingUnit.brew.brew (CoffeeSelection.FILTER_COFFEE, groundCoffee, this.config.getQuantityWater()); } public void addGroundCoffee(CoffeeSelection flood, GroundCoffee newCoffee) throws CoffeeException { GroundCoffee existingCoffee = this.groundCoffee.get(sel); if (existingCoffee != nu) { if (existingCoffee.getName().equal newCoffee.getName())) { existing existingCoffee.setQuantity(existingCoffee.getQuantity() + newCoffee.getQuantity()); } else { throw new CoffeeException (Only one type of coffee supported for each CoffeeSelection.); } else { this.groundCoffee.put(sel, newCoffee); } } } } Premium kahve makinesiAma FilterCoffeeMachine arayüzü uygulamayan yeni, daha gelişmiş bir kahve makinesi alırsanız ne olur?genel sınıf PremiumCoffeeMachine { özel Harita<CoffeeSelection, configuration=> configMap; özel Harita<CoffeeSelection, coffeebean=> fasulye; özel Grinder öğütücü; özel BrewingUnit brewingUnit; kamu PremiumCoffeeMachine (Harita<CoffeeSelection, coffeebean=> fasulye) { this.beans = fasulye; this.grinder = new Grinder(); this.brewingUnit = new BrewingUnit(); this.configMap = new HashMap <> (); this.configMap.put(CoffeeSelection.FILTER_COFFEE, new Configuration(30, 480)); this.configMap.put(CoffeeSelection.ESPRESSO, new Yapılandırma(8, 28)); } public Coffee brewCoffee(CoffeeSelection selection) atar CoffeeException { switch (selection) { case FILTER_COFFEE brewFilterCoffee(); case ESPRESSO: return brewEspresso(); default: throw new CoffeeException(CoffeeSelection + selection + desteklenmez); } } } özel Kahve brewEspresso() { Configuration config = configMap.get(CoffeeSelection.ESPRESSO); // grind kahve çekirdekleri GroundCoffee groundCoffee = this.grinder.grind(this.beans.get(CoffeeSelection.ESPRESSO), config.getQuantityCoffee()); // brew a espresso return this.brewingUnit.brew(CoffeeSelection.ESPRESSO, groundCoffee, config.getQuantityWater()); } private Coffee brewFilterCoffee()) } Private Coffee brewFilterCoffee() { Configuration config = configMap.get(CoffeeSelection.FILTER_COFFEE); // grind the coffee beans GroundCoffee groundCoffee = this.grinder.grind(this.beans.get(CoffeeSelection.FILTER_COFFEE) , config.getQuantityCoffee()); brew a filter coffee return this.brewingUnit.brew(CoffeeSelection.FILTER_COFFEE, groundCoffee, config.getWaterQuantity()); } public invalid addCoffeeBeans(CoffeeSelection flood, CoffeeBean newBeans) KahveException { CoffeeBean = atar</CoffeeSelection,> </CoffeeSelection,> </CoffeeSelection,> </CoffeeSelection,> </CoffeeSelection,>

</CoffeeSelection,> if (existingBeans != null) { if (existingBeans.getName().equals(newBeans.getName()) { existingBeans.setQuantity(existingBeans.getQuantity() + newBeans.getQuantity()); else { throw new CoffeeException(Coffee supported for each CoffeeSelection is just a type.); } else { this.beans.put(sel, newBeans); } } } } You can use PremiumCoffeeMachine's CoffeeException method to prepare filter coffee or espresso. As you can see, the method has the same name as defined by the FilterCoffeeMachine interface, but the method signature is incompatible. Waits for a parameter and notes an exception. PremiumCoffeeMachine represents a coffee machine, but does not implement the FilterCoffeeMachine interface. Therefore, you cannot use it with FilterCoffeeApp. I will not change the class to implement the necessary interface. It is often not possible to modify existing classes because they are implemented by a different team or classes are used in other projects that do not have the required interface. I don't want to change the FilterCoffeeMachine interface either. BasicCoffeeMachine implements this interface and I need to change this class when I change the interface. In such cases, it is better to apply an adapter pattern. By applying the adapter, you enable your FilterCoffeeApp to use the coffee machine by introducing an adapter class that implements the FilterCoffeeMachine interface and envelops the PremiumCoffeeMachine class. In this example, the adapter class must perform two missions:filterCoffeeMachine interface. The BrewCoffee method should close the gap between the brewCoffee method defined by the interface and the brewCoffee method implemented by the PremiumCoffeeMachine class. The interface and existing class are not very different. This simplies the implementation of the adapter class.public class FilterCoffeeAdapter FilterCoffeeMachine { custom Logger log = Logger.getLogger (FilterCoffeeAdapter.class.getSimpleName()); implements custom PremiumCoffeeMachine machine; public FilterCoffeeAdapter(PremiumCoffeeMachine machine) { this.machine = machine; } @Override public Coffee brewCoffee() { try { return machine.brewCoffee (CoffeeSelection.FILTER_COFFEE); } catch (CoffeeException e) { log.severe(e.toString()); return null; } } } As you can see in the code snippet, the FilterCoffeeAdapter class implements the FilterCoffeeMachine interface and waits for a PremiumCoffeeMachine object as a constructor parameter. It holds this object in a special area so that it can use it in the brewCoffee method. The implementation of the BrewCoffee method is critical and, for the most adapter classes, the most difficult part. In this case, the PremiumCoffeeMachine class provides a method that you can call to perform the task. But it is more flexible and requires a CoffeeSelection enum value to define the coffee that will produce. The Method of the PremiumCoffeeMachine class FILTER_COFFEE CoffeeException if it is called with a different CoffeeSelection value than the ESPRESSO and ESPRESSO. The brewCoffee method of the FilterCoffeeMachine interface does not report this exception, and you must process it within the method implementation. In this example, there is no perfect way to do this. If you can write a log message and return null, as I did in the code snippet, or RuntimeException.In from your application, you may have better ways to deal with the exception. You can re-attempt or trigger a different business process. This will make your application more robust and will better implement your adapter. The SummaryThe Adapter Pattern is a common pattern in object-oriented programming languages. Similar to adapters in the physical world, you implement a class that closes the gap between the expected interface and an existing class. This allows you to re-use an existing class that does not implement the required interface and use the functionality of multiple classes that would otherwise be incompatible. One advantage of The Adapter Pattern is that you do not need to modify the existing class or interface. By introducing a new class that performs adapters between the interface and the class, you avoid making any changes to existing code. This limits the scope of your changes to your software component and prevents any changes and side effects to other components or applications. Adapter Pattern Dependency Inversion ensures that the design policy is applied. If you are already unfamiliar, I recommend reading about different SOLID design principles. I wrote a series of articles describing five: APM, with server health metrics and error log integration, improve application performance with Stackify Retrace. Try a free two-week trial today