SOLID principles

Q1.

1. It violates ***DEPENDENCY INVERSION*** principle.
2. The principle states that High level modules should not depend on low level modules, both should depend on abstractions. In our example we have Employer higher level class is depended on lower level classes. If you want to add different kind of worker other than Hourly and Second you have to change the higher level class. In short if you have more different workers you have to change code in Employer class constructor and methods.
3. So we created an interface Payable having calculatepay method and let the lower level classes implement it. Then in high level class constructor we are passing array of interface object which will take any form of class that implements Payable interface and an integer array containing number of employees. In this way if we want to add more type of workers we have to just create class that implements Payable interface and provide implementation of calculatepay method in the class created and add it in array while calling Employer method. We don't have to change the higher level class in this manner.

Driver Code

```
        Employer er=new Employer(new Payable[]{new HourlyWorker(),new
SalaryWorker()},new Integer[]{5,5});
        er.outputWageCostsForAllStaff(20);
```

Q2.

1. It violates ***INTERFACE SEGREGATION*** principle.
2. This principle states that no client should be forced to depend on methods it does not use. In our example we see that we have one interface that has all methods and the classes implements that interface. But we clearly see that Book.java class forcefully implements getCastList and getPlayTime methods which it should not do vice versa for the DVD.java class.
3. So we created three separate interface and we know that java class can implement multiple interface. We have segregated the interface and let class implement it based on the methods they want to implement. There is no forceful implementation of any method in this way and we do not violate this principle.

Q3.

1. It violates ***SINGLE RESPONSIBILITY*** principle.
2. This principle states that Every class should have a single responsibility. There should never be more than one reason for a class to change. In our example PrintReport.java class does all the task of creating, sending and emailing. So the single class has three tasks to perform which is contrary to our principle.

3. So we created three classes one to create Report, second to send to printer and third to just send to email. So every class has only one reason to change. This makes all the classes having only one work and does not violate this principle.

Q4.

1. It violates ***LISKOV SUBSTITUTION*** principle.
2. The principle states that if a program module is using a Base class, then the reference to the Base class can be replaced with a Derived class without affecting the functionality of the program module. In the given example USDollarAccount does not have getBalance method and when it extends Base class Bankaccount it can access it's method. So it is violating Liskov Substitution principle. Moreover if you change balance variable of sub class it also changes the balance variable of BankAccount class when you call getBalance method from BankAccount object.
3. We created an Abstract class Account and added float balance, abstract credit and debit methods to it. Then we let the USDollarAccount and BankAccount extend the abstract Account class. In this manner every class has it's own implementation of debit and credit methods and we let the BankAccount implement Balance interface for getBalance method which USDollarAccount does not have it.

Q5.

1. It violates ***DEPENDENCY INVERSION*** principle.
2. The principle states that High level modules should not depend on low level modules, both should depend on abstractions. In our example we have CountryGDPReport higher level class that depended on lower level classes. If you want to add different kind of countries other than Canada and Mexico you have to change the higher level class. In short if you have more different countries you have to change code in CountryGDPReport class.
3. So we created an interface Report having PrintGDPReport method and let the lower level classes implement it. Then in high level class constructor we are passing interface object which will take any form of class that implements the interface. In this way if we want to add more type of countries we have to just create new class that implements Report interface and provide implementation of PrintGDPReport method in the class. There is method in CountryGDPReport which calls PrintGDPReport method of respective classes. We don't have to change the higher level class CountryGDPReport in this manner.

Driver Code

```
Report canada=new Canada();
Report mexico=new Mexico();
CountryGDPReport c1=new CountryGDPReport(canada);
CountryGDPReport c2=new CountryGDPReport(mexico);
c1.PrintCountryGDPReport();
c2.PrintCountryGDPReport();
```

Q6.

1. It violates ***SINGLE RESPONSIBILITY*** principle.
2. This principle states that Every class should have a single responsibility. There should never be more than one reason for a class to change. In our example PiggyBank.java class does all the task of loading, adding and saving the money. So the single class has three tasks to perform which is contrary to our principle.
3. So we created three classes one to load money, second to add money and third to just save the money. So every class has only one reason to change. This makes all the classes perform only one work and does not violate this principle.

Q7.

1. It violates ***INTERFACE SEGREGATION*** principle
2. This principle states that no client should be forced to depend on methods it does not use. In our example we see that we have one interface that has all methods and the all classes implements that interface. But we clearly see that AcquaticInsect.java class forcefully implements Fly method which it should not.
3. So we created three separate interface and we know that in java class can implement multiple interface. We have segregated the interface and let class implement the respective interfaces based on the methods they want to implement. There is no forceful implementation of any method in this way and we do not violate this principle.