1. Which of the following is a better way of writing code

A)

```
for(Vehicle vehicle : vehicles) {
    switch(vehicle.getType()) {
        case CAR:
            vehicle.lock();
            vehicle.go();
            break;
        case SHIP:
            vehicle.balance();
            vehicle.swim();
            break;
        case AIRPLANE:
            vehicle.go();
            vehicle.fly();
            break;
        case TANK:
            vehicle.move();
            vehicle.stop();
            vehicle.fire();
            break;
    }
    vehicle.stop();
}
```

B)

```
do(Car v){
    vehicle.lock();
    vehicle.go();
}
do(Ship v){
     vehicle.balance();
     vehicle.swim();
}
do(Airplane v){
     vehicle.go();
     vehicle.fly();
}
do(Tank v){
     vehicle.move();
     vehicle.stop();
     vehicle.fire();
}

execute(List<Vehicle> vehicles){
    for(Vehicle vehicle : vehicles) {
        do(vehicle);
        vehicle.stop();
```

```
    }
}
```

C)

```
interface Vehicle{
  do();
  stop();
}
class Car implements Vehicle{
  do(){
     lock();
     go();
  }
  ...
}
class Ship implements Vehicle{
  do(){
     balance();
     swim();
  }
  ...
}
class Airplane implements Vehicle{
  do(){
     go();
     fly();
  }
  ...
}
class Tank implements Vehicle{
  do(){
     move();
     stop();
     fire();
  }
  ...
}

execute(List<Vehicle> vehicles){
    for(Vehicle vehicle : vehicles) {
        vehicle.do();
        vehicle.stop();
    }
}
```

2. What are the code smells in the following code

```
class StatisticsReport
{
    protected string document;

    public string getData(string format)
    {
        switch(format) {
            case "csv":
                lines = [];
                foreach (this->document as row) {
                    lines = implode(",", row);
                }
                return implode("\n", lines);

            case "json":
                json="";

                //some logic for formating as json ...
                return word

            case "html":
                html = "";

                //  some logic for formating as HTML ...
                return html;

             case "xml":
                xml = "";

                //  some logic for formating as xml ...
                return xml;
        }
    }
}
```

A) Magic strings
B) OCP issue
C) Cyclomatic complexity
D) God Class

3. How to reduce the Complexity of the following code

```
double getSalary(int empCode)
{
     Repostory repo = new Repository();
     res = repo.connect();
     if(res == true){
          res = repo.authenticate();
          if(res == true){
               Emp emp = rep.get(empCode);
               if(emp != null){
                    return emp.getSalary();
               }else{
                    return 0;
               }
          }else{
               return 0;
          }
     }else{
          return 0;
     }
}
```

A) using interface
B) Using Exception Handling
C) Combining if conditions

4. Which of the following is a better way of writing code

A)

```
class Tax
{
  public static double compute(int taxType, double amount){
      switch(taxType)
      {
     case 1:
          if(amount > 1000)
               amount += amount * 0.05;
          break;
     case 2:
          amount += amount * 0.025 + 500;
          break;
     case 3:
          if(amount < 1000)
               amount += (amount- 5000) * 0.3;
          else
               amount += (amount- 5000) * 0.4;
          break;
      }
      return amount;
  }

}

public class Invoice {
     private int taxType;


     public int getTaxType() {
          return taxType;
     }

     public void setTaxType(int taxType) {
          this.taxType = taxType;
     }

     public double getTotal()
     {

          double amount = getSubtotal();
             amount += Tax.compute(taxType,amount);
          return amount;
     }
     double getSubtotal()
     {
          ...
     }

}
```

B)

```
interface Tax
{
     double compute(double amount);
}
public class Invoice {
     private Tax tax= new TaxImp();

     public double getTotal()
     {
          double amount = getSubtotal();
             amount += tax.Compute(amount);
          return amount;
     }
     double getSubtotal(){
          ...
     }
}
class TaxImp  implements Tax
{
   int taxType;

  public TaxImp(int taxType)
  {
        this.taxType = taxType;
  }
  public double compute(double amount){
      double taxAmount =0;


      switch(taxType)
      {
     case 1:
          if(amount > 1000)
               amount += amount * 0.05;
          break;
     case 2:
          amount += amount * 0.025 + 500;
          break;
     case 3:
          if(amount < 1000)
               amount += (amount- 5000) * 0.3;
          else
               amount += (amount- 5000) * 0.4;
          break;
      }
      return taxAmount;
  }

}
```

C)

```
public class Invoice {
     private int taxType;

     void setTaxType(int taxType) {
          this.taxType = taxType;
     }

     public double getTotal(){
          double amount = getSubtotal();
          switch(taxType)
          {
               case 1:
                    if(amount > 1000)
                         amount += amount * 0.05;
                    break;
               case 2:
                    amount += amount * 0.025 + 500;
                    break;
               case 3:
                    if(amount < 1000)
                         amount += (amount- 5000) * 0.3;
                    else
                         amount += (amount- 5000) * 0.4;
                    break;
          }
          return amount;
     }
     double getSubtotal()
     {
          ..
     }
}
```

D)

```
interface Tax
{
     double compute(double amount);
}
public class Invoice {
     private Tax tax;

     public void setTaxType(Tax tax) {
          this.tax = tax;
     }
     public double getTotal()
     {
          double amount = getSubtotal();
             amount += tax.Compute(amount);
```

```
          return amount;
     }
     double getSubtotal(){
          ...
     }
}
class KST  implements Tax
{
  public double compute(double amount){
      if(amount > 5000)
          return amount * 0.05;
      else
          return amount * 0.05 + 200;
  }
}
class GST  implements Tax
{
   public double compute(double amount){
     return amount * 0.025 + 500;
  }
}
class CST  implements Tax
{
    public double compute(double amount){
     return (amount- 5000) * 0.3;
  }
}
```

5. An Account class is defined as below

```
public class Account {
     double balance;

     public boolean withDraw(double amount)
     {
          if(amount > balance)
               return false;

          balance -= amount;
          return true;
     }
     public void deposit(double amount)
     {
          balance += amount;
     }
}
```

Which of the following is a better code for handling undo operation in Accounts

A)

```
class Operation
{
  public int type;
  public double amount;
}

class AccountService
{
  Stack<Operation> stack = new Stack<Operation>();
  Account acc = new Account();

  public void withdraw(double amount){
    acc.withdraw(amount);
    Operation op = new Operation(1,amount);
    stack.push(op);
  }

  public void deposit(double amount){
    acc.deposit(amount);
    Operation op = new Operation(2,amount);
    stack.push(op)
  }

  public void undo(){
    Operation op = stack.pop();
    if(op.Type == 1)
      acc.deposit(op.amount);
```

```
    if(op.Type == 2)
      acc.withdraw(op.amount);

  }
}
```

B)

```
class AccountService
{
  Stack<Lamda> stack = new Stack<Lamda>();
  Account acc = new Account();

  public void withdraw(double amount){
    acc.withdraw(amount);
    stack.push(()->acc.deposit(amount));
  }
  public void deposit(double amount){
    acc.deposit(amount);
    stack.push(()->acc.withdraw(amount));
  }
  public void undo(){
    Lamda fun = stack.pop();
    fun();
  }
}
```

C)

```
interface Operation{
  undo(Account acc);
}
class DepositOp implements Operation
   double amount;
   public void undo(Account acc){
     acc.withdraw(amount);
   }
}
class WithdrawOp implements Operation
   double amount;
   public void undo(Account acc){
     acc.deposit(amount);
   }
}
class AccountService
{
  Stack<Operation> stack = new Stack<Operation>();
  Account acc = new Account();

  public void withdraw(double amount){
    acc.withdraw(amount);
    Operation op = new WinthDrawOpr(amount);
```

```
    stack.push(op);
  }
  public void deposit(double amount){
    acc.deposit(amount);
    Operation op = new DepositOpr(amount);
    stack.push(op)
  }
  public void undo(){
    Operation op = stack.pop();
    op.undo(acc);
  }
}
```

6. What is the output of the following code

```
class CA{}
class CB extends CA{}
class CC extends CB{}
class Util{
  void f(CA){} //1
  void f(CB){} //2
  void f(CC){} //3
}

void main()
{
     CC c = new CC();
     CB b = c;
     CA a = c;

     Util util = new Util();
     uitl.f(a);
     uitl.f(b);
     uitl.f(c);
}
```

A) 1,1,1.
B) 1,2,3.
C) 3,3,3.

7. What is the output of the following code

```
class CA{
}
class CB exends CA{
}
class CC exends CA{
}
class CX{
  void f(CA) {}//1
  void f(CB) {}//2
  void f(CC) {}//3
}
class CY. extends CX {
  void f(CA) {}//4
  void f(CB) {}//5
  void f(CC) {}//6
}
class CZ. extends CX {
  void f(CA) {}//7
  void f(CB) {}//8
  void f(CC) {}//9
}
do(CA a, CX x){
    x.f(a);
}
do(new CC, new CZ);
```

A) 1
B) 9
C) 7

8. Consider the following classes in a Gaming application.

```
interface GameObject{}
class Ship extends GameObject{}
class Station extends GameObject{}
class Commet extends GameObject{}
class Aestroid extends GameObject{}
```

Which of the following is a better code for processing collusion between any 2 game objects.

A)

```
class Handler
{
     public void Collide(GameObject go1,GameObject go2)
     {
          if(type(o1)==type(Ship) && type(o2)==type(Station))
          {
            //logic1
          }
           if(type(o1)==type(Station) && type(o2)==type(Aestroid))
          {
            //logic2
          }
           if(type(o1)==type(Ship) && type(o2)==type(Commet))
          {
            //logic3
          }
     }
}
```

B)

```
class Handler
{
     public void Collide(Ship go1,Station go2) {}
     public void Collide(Aestroid go1,Station go2) {}
     public void Collide(Commet go1,Station go2) {}
     public void Collide(Commet go1,Ship go2) {}
     ....
}

     public void Invoke(GameObject go1,GameObject go2)
     {
        Handler handler = new Handler();
        handler.Collide(go1,go2);
     }
```

C)

```
class Ship extends GO{
```

```
  void Collide(GameObject go2){
    //this is ship
    if(go2 is instanceof(Station))
      ...logic of ship with station
    if(go2 is instanceof(Commet))
      ...logic of ship with commet
    if(go2 is instanceof(Aestroid))
      ...logic of ship with aestroid
  }
}

class Handler
{
     public void Collide(GameObject go1,GameObject go2)
     {
          go1.Collide(go2);
     }
}
```

D)

```
class Ship extends GO{
  void Collide(GameObject go2){
    go2.collide(this);
  }
  void Collide(Station go2){
    // logic to  collide ship and station
  }
  void Collide(Commet go2){
    // logic to  collide ship and Commet
  }
  void Collide(Aestroid go2){
    // logic to  collide ship and Aestroid
  }
  void Collide(Ship go2){
    // logic to  collide ship and Ship
  }
}

class Station extends GO{
  void Collide(GameObject go2){
    go2.collide(this);
  }
  void Collide(Ship go2){
    // logic to  collide station and ship
  }
  void Collide(Station go2){
    // logic to  collide station and station
  }
  void Collide(Commet go2){
    // logic to  collide station and Commet
  }
```

```
  void Collide(Aestroid go2){
    // logic to  collide station and Aestroid
  }

}

class Handler
{
     public void Collide(GameObject go1,GameObject go2)
     {
          go1.Collide(go2);
     }
}
```

9. What is the output of the following code

```
class CA{
    void invoke(Util u){  u.do(this);  }
}
class CB extends CA{
   void invoke(Util u){  u.do(this);  }
}
class CC extends CB{
```

```
  void invoke(Util u){  u.do(this);  }
}
class Util{
  void do(CA){} //1
  void do(CB){} //2
  void do(CC){} //3
}


CC c = new CC();
CB b = c;
CA a = c;

Util util = new Util();
a.invoke(util);
b.invoke(util);
c.invoke(util);
```

A) 1,1,1.
B) 1,2,3.
C) 3,3,3.

10. Suppose Queue class and Stack class want to reuse the code in List class, which is the better way of reusing the code.

```
public class List {
     //collection
     public void add(Object item,int index ) {
          //logic to add item into collection
     }
     public void remove(int index ) {
          //logic to remove item from collection
```

```
     }
}
```

A)

```
public class Queue extends List {
     public void enqueue(Object item) {
          //add to the list methods
     }
     public void dequeue() {
          //remove first item from the list
     }
}
public class Stack extends List {
     public void push(Object item) {
          //add to the list methods
     }
     public void pop() {
          //remove last item from the list
     }
}
```

B)

```
public class Queue {
     List ref;
  public void enqueue(Object item) {
          //delegate to List methods
     }
     public void dequeue() {
          //delegate to List methods
     }
}
public class Stack {
     List ref;

  public void push(Object item) {
          //delegate to List methods
     }
     public void pop() {
          //delegate to List methods
}
```