Q1.b

Polymorphism

Polymorphism refers to the ability of the design that enables an object or a method to take on multiple forms.

When we use an instance of the child class to initialize or assign to a parent class reference, it is polymorphism. This type of polymorphism is also known as subtype polymorphism, as the child class will be a subtype of the parent class. The two other types of polymorphism are - Function overriding (runtime polymorphism) and Function overloading (compile-time polymorphism).

Good use:

The below example is one of the good uses of Polymorphism. getPrice() is an empty method in the class Car. The child class HondaCrv implements it as the price of the car is associated with a particular model of the class. Even if this function is defined in the parent class Car, it can be overridden by any child class that extends it.

```
class Car {
    public void getPrice() {   }
}

class HondaCrv extends Car {
    public void getPrice() {
        // getPrice() definition
    }
}
```

Bad use:

Suppose the requirement is that we want to restrict the child class from overriding some methods of parent class. We cannot achieve this because of the runtime polymorphism capability available in Object-oriented programming (although we could use the “final” keyword to satisfy our requirement here). So, in this case, polymorphism is not helpful here and is an example of bad usage.

```
class A {
    public void method1() {
        // Method definition
    }
}

class B extends A {
    public void method1() {
        // Method definition
    }
    // Here method1() is being overridden by the child class B
}
```

Q1.c

Cohesion

Cohesion indicates the degree to which a class or function has a single, well-focused purpose. If a class or method has low cohesion, then it means that it is performing multiple tasks. On the other hand, if it has high cohesion, then it means that it has one single and well-focused responsibility. If the cohesiveness is higher then it is a better design.

Bad use of cohesion:

The below code snippet shows a class with less cohesiveness which does multiple unrelated tasks. Here, HondaCrv class maintains a list of the customers who bought it. Logically, the task of maintaining customer names is not the responsibility of HondaCrv class. So this class has low cohesion.

```
class HondaCrv {
    private List<String> customerNames;
    public HondaCrv() {
        customerNames = new ArrayList<>();
    }
    public List<String> getCustomers() {
        return customerNames;
    }
    public void addCustomer(String customerName) {
        customerNames.add(customerName);
    }
    // other variables and methods definition of HondaCrv class
}
```

Good use of cohesion:

One of the good use of cohesion is where the class has high cohesion and has only one and well-focused responsibility. The ‘HondaCrv’ class has the behavior and properties related to Honda CRV car only. The HondaCrvCustomer class has the task of maintaining a list of customers who bought Honda CRV. The Customer class is a more high-level class that maintains the details of a customer. All these classes have only one clear responsibility and have high cohesion.

```
class HondaCrv {
    public HondaCrv() {
        // initializations done here
    }
    // other variables and methods definition of HondaCrv class
}

class Customer {
    private String name;
    public Customer() {
        // initializations done here
    }
    public void setName(String name) {  this.name = name;  }
}

class HondaCrvCustomers {
    private List<Customer> customers;
    public HondaCrvCustomers() {
        // initializations done here
    }
    public List<Customer> getCustomers() {
        return customers;
    }
    public void addCustomer(Customer customer) {
        customers.add(customer);
    }
}
```