Understanding Encapsulation Using Examples

In this lesson, you will get a firmer understanding of encapsulation in C# with the help of examples.

WE'LL COVER THE FOLLOWING

- A Bad Implementation
- A Good Implementation

As discussed earlier, encapsulation refers to the concept of binding **data and the methods operating on that data** in a single unit also called a class.

The goal is to prevent this bound data from any unwanted access by the code outside this class. Let's continue working with our example of the `VendingMachine` class to understand the concept of encapsulation.

We know that vending machines collect money from the customers and in return provide them their desired product. This money collection is done with the help of a money collector module installed on the machine. Let's simulate this module in the form of a class.

A very basic `MoneyCollector` class' object should be able to perform the following:

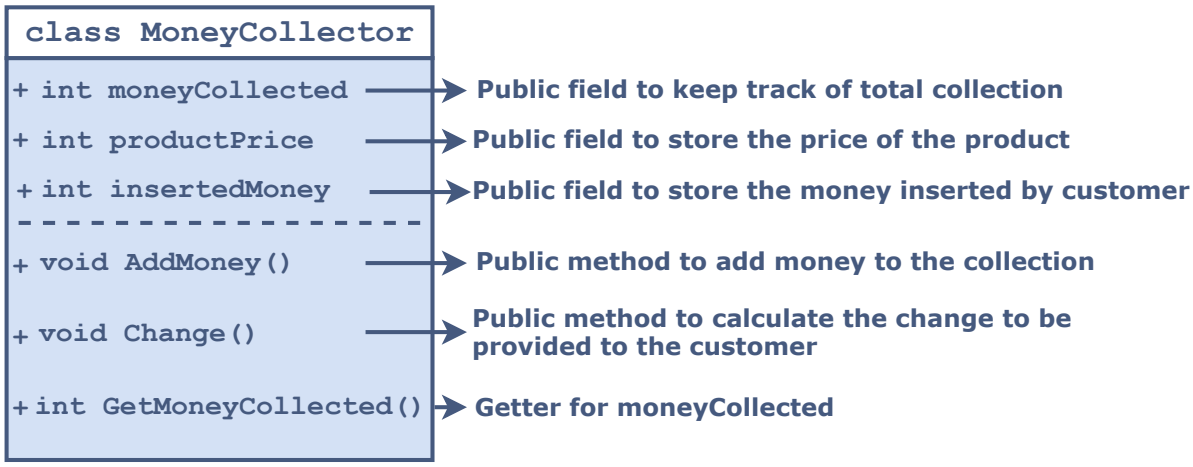
- Keep track of the total sales.
- Add the collected money to the total.
- Provide the change to the user in case the product was of a lesser price than the inserted amount of money.

For the sake of simplicity, we will assume that we have only one product with a fixed price.

A money collector module

A Bad Implementation

Now it is time to implement the above discussed `MoneyCollector` class.



A bad example of encapsulation

The code according to the above illustration is given below:

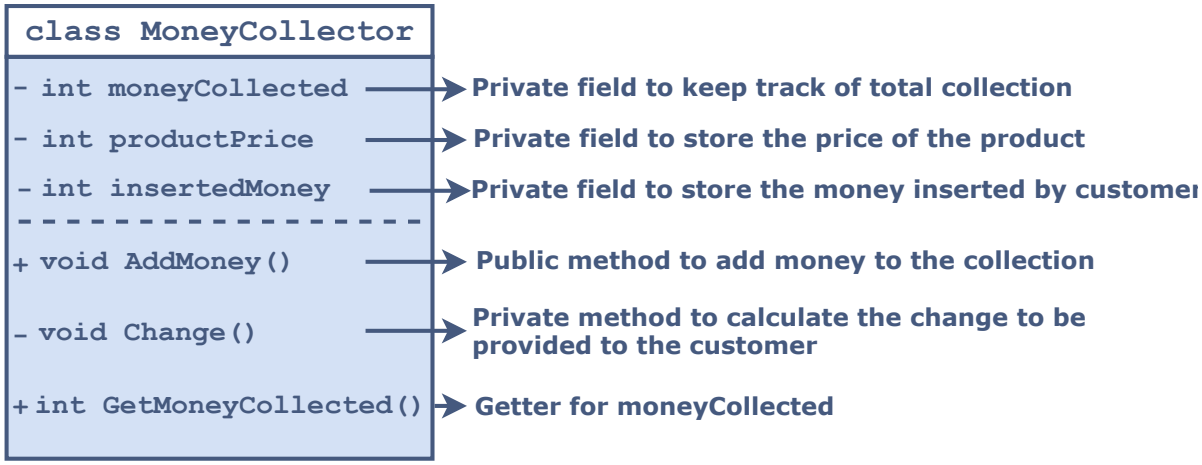
```
// MoneyCollector Class
class MoneyCollector {

  // Public Fields
  public int moneyCollected;
  public int productPrice;
  public int insertedMoney;
```

```
  public int insertedMoney;


  // Parameter-less Constructor to intialize the money collector object
  public MoneyCollector() {
    this.moneyCollected = 0;
    this.productPrice = 2; // Let's fix the product price to 2$
    this.insertedMoney = 0;
  }

  public void AddMoney(int money) { // Method to add money to collection
    this.insertedMoney = money;
    if (this.insertedMoney >= 0) {  // Check if the customer inserted valid money
      Console.WriteLine("You inserted {0}$",this.insertedMoney);
      this.Change(); // Call the change method to provide change
    }
    else Console.WriteLine("Invalid Insertion");

    this.insertedMoney = 0;
  }

  public void Change() { //method to provide change
    if (this.insertedMoney >= this.productPrice) { //check if any change
      int change = this.insertedMoney - this.productPrice; //calculate change
      // product sold so add its price to collected money
      this.moneyCollected += this.productPrice;
      Console.WriteLine("Your change is: {0}$", change);
    }
    else {
      Console.WriteLine("You didn't insert sufficient money!");
      // the transaction was not successfull so return back the money
      Console.WriteLine("Your change is: {0}$",this.insertedMoney);
    }
  }

  public int GetMoneyCollected() { // Getter to moneyCollected
    return this.moneyCollected;
  }

}

class Demo {

  public static void Main(string[] args) {
    // Create a new money collector object
    var moneyCollector = new MoneyCollector();
    // 3 Customers purchase products
    moneyCollector.AddMoney(2);
    moneyCollector.AddMoney(5);
    moneyCollector.AddMoney(7);
    // getting the collected as 3 products sold it should be 2*3 = 6
    Console.WriteLine("Total collection till now is: {0}$",moneyCollector.GetMoneyCollected()

    //Let's try to corrupt collection
    moneyCollector.moneyCollected = 20;
    Console.WriteLine("Total collection till now is: {0}$",moneyCollector.GetMoneyCollected()
    //The collection was public so we easily changed its value
    //THIS SHOULD NOT HAVE HAPPENED!

  }

}
```

In the above coding example, we can observe that **anyone** can *access and change* the `moneyCollected` field directly from the `Main()` method. This is **dangerous** because there is no encapsulation of the `moneyCollected` field. The above code was a poor design choice. The `Change()` method should be declared private because there is no need to call it independently from the `Main()`.

A Good Implementation

Let's move on to a good convention for implementing the `MoneyCollector` class!

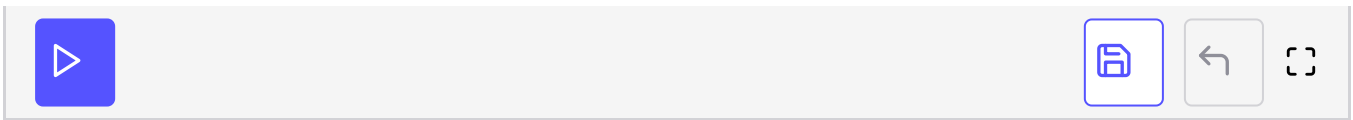


A good example of encapsulation

The code according to the above illustration is given below:

```
// MoneyCollector Class
class MoneyCollector {

  // Public Fields
  private int moneyCollected;
  private int productPrice;
  private int insertedMoney;


  // Parameter-less Constructor to intialize the money collector object
  public MoneyCollector() {
    this.moneyCollected = 0;
    this.productPrice = 2; // Let's fix the product price to 2$
    this.insertedMoney = 0;
  }

  public void AddMoney(int money) { // Method to add money to collection
```

```
  public void AddMoney(int money) { // Method to add money to collection
    this.insertedMoney = money;
    if (this.insertedMoney >= 0) {  // Check if the customer inserted valid money

      Console.WriteLine("You inserted {0}$",this.insertedMoney);
      this.Change(); // Call the change method to provide change
    }
    else Console.WriteLine("Invalid Insertion");
    // as a good practice set inserted to 0 at the end of transaction
    this.insertedMoney = 0;
  }

  private void Change() { //method to provide change
    if (this.insertedMoney >= this.productPrice) { //check if any change
      int change = this.insertedMoney-this.productPrice; //calculate change
      this.moneyCollected += this.productPrice; // Add money to total collection
      Console.WriteLine("Your change is: {0}$",change);
    }
    else {
      Console.WriteLine("You didn't insert sufficient money!");
      // Return whatever the user entered
      Console.WriteLine("Your change is: {0}$",this.insertedMoney);
    }
  }

  public int GetMoneyCollected() { // Getter to moneyCollected
    return this.moneyCollected;
  }

}

class Demo {

  public static void Main(string[] args) {
    // Create a new money collector object
    var moneyCollector = new MoneyCollector();
    // 3 Customers purchase products
    moneyCollector.AddMoney(3);
    moneyCollector.AddMoney(5);
    moneyCollector.AddMoney(7);
    // getting the collected as 3 products sold it should be 2*3 = 6
    Console.WriteLine("Total collection till now is: {0}$",moneyCollector.GetMoneyCollected()

    // Uncommenting the below line will now cause an error
    //moneyCollector.moneyCollected = 20;

  }

}
```



In the above example, all the fields are declared `private`.

As a rule of thumb, in a class, all the member variables should be declared `private` and to access and operate on that data, `public` methods like *getters, setters and custom methods* should be implemented. We should look at the

ways in which we would interact with a class' objects. For instance, what are the interactions that one can have with a vending machine? All the features should become public methods while other helper methods can be declared private.

This is the concept of encapsulation. All the fields containing data are private and the methods which provide an interface to access those fields are public.

Adding to this, a good implementation of a class is, when it serves only a single purpose, i.e., a class should always be implemented in such a way that it is responsible for a single task. To perform multiple tasks, multiple classes should be implemented rather than implementing everything in a single class.

Now let's test your understanding of encapsulation with the help of a quick quiz!