CHANGEBOT

Stephen Jones - Coder

Treyveon Craig - Spotter

OVERVIEW

Back-end software
for a self checkout kiosk

Transaction logging

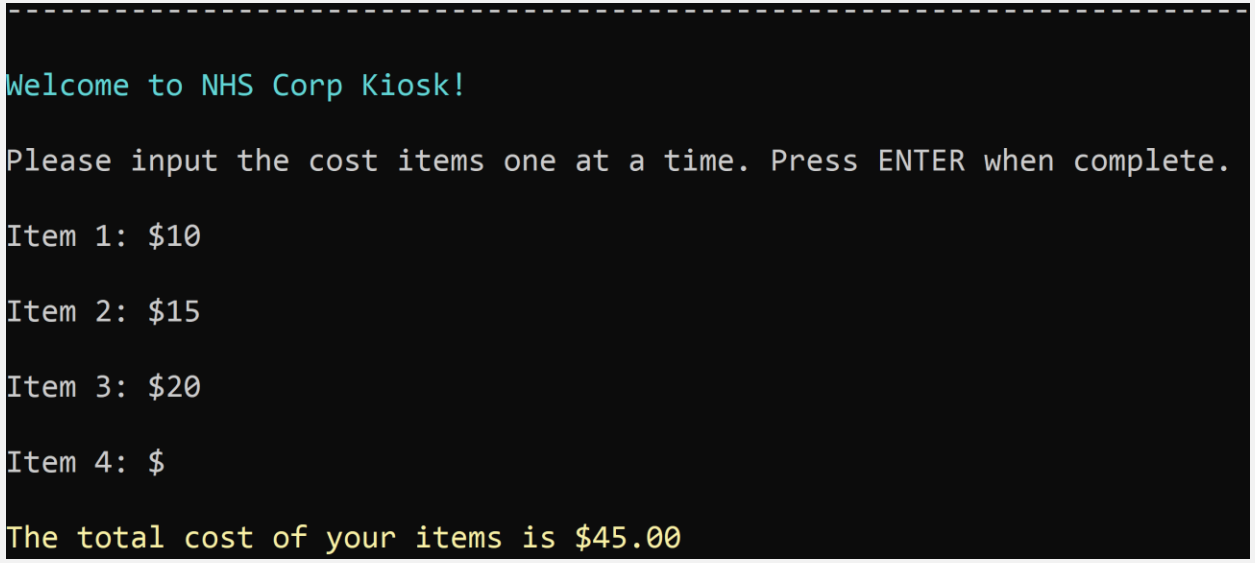
FIRST EDITION – CASH ONLY

```
#region GLOBAL DRAWER VALUES, AMOUNTS AND NAMES
//global drawer, names and real values of currency
static readonly decimal[] real_values    = {100.00m, 50.00m, 20.00m, 10.00m, 5.00m, 2.00m, 1.00m, 1.00m, 0.50m, 0.25m, 0.10m, 0.05m, 0.01m};
static readonly int[]     cash_values    = {10000,5000,2000,1000,500,200,100,100,50,25,10,5,1};
static int[]              cash_drawer    = {0, 2, 5, 3, 5, 0, 25, 0, 5, 40, 50, 40, 50};
static readonly string[]  cash_names     = {"$100 bill", "$50 bill", "$20 bill", "$10 bill", "$5 bill", "$2 bill", "$1 bill",
                                            "$1 coin", "$0.50 coin", "$0.25 coin", "$0.10 coin", "$0.05 coin", "$0.01 coin"};

#endregion
```

- Small beginnings – only cash
- Money drawer system, struct array
- System persists throughout the program's evolution

INPUT ITEM COSTS



Above: User interface

Right: Code snippet from input item function

```
static decimal  ScanItems() {
    decimal total_cost = 0.0m;
    string  input_item;
    bool    correct_input;
    int     count      = 1;

    //welcome, line window with '-'
    LineWindow('-');
    ColorChangeCyan  ("Welcome to NHS Corp Kiosk!");
    Console.WriteLine("\nPlease input the cost items one at a time. Press ENTER when complete.");

    //loops till input is empty
    do {
        do {
            //input validation loop, only accept numbers and '.'
            correct_input  = true;

            //get input as string to check if correct input
            input_item     = PromptString($"\nItem {count}: $");

            //CHECK FOR INCORRECT CHARS, AKA ANYTHING OTHER THAN NUMBERS OR '.'
            foreach (char character in input_item) {
                if (character > 57 || character < 48 && character != 46) {
                    correct_input = false;
                }
            }

            //DISPLAY ERROR IF INCORRECT INPUT
            if(correct_input == false){
                ColorChangeRed   ("\nItem not accepted.");
                Console.WriteLine("Please re-enter item without comma.");
            }

        } while (!correct_input);

        //IF INPUT IS NOT EMPTY, ADD ITEM PRICE TO TOTAL COST
        if (input_item != "") {

            //CONVERT TO DECIMAL AND ADD ITEM COST TO TOTAL COST
            total_cost += decimal.Parse(input_item);
        }

        //COUNT ITEM NUMBER
        count++;
    }while (input_item != "");

    ColorChangeYellow($"\nThe total cost of your items is {total_cost:C}");

    return total_cost;
}
```

INPUT PAYMENT

```
Enter current legal US tender amounts as bills/coins one at a time: $9.99

ERROR: Please enter payments one at a time as current legal US tender bills/coins.
```

Above: User interface

Right: Input validation loop

```
decimal  PaymentCash     (decimal total_cost, decimal cash_payment, ref decimal total_cash_payment) {
imal input_payment;
ing  input;

NPUT VALIDATION, ONLY ACCEPT US LEGAL NOTE/COIN VALUES (EX: 1.00, 0.50, 20.00 ETC...)
{
 bool correct_value     = false;
 bool coin              = false;
 bool correct_input;

  do {
     //INPUT VALIDATION LOOP, ONLY ACCEPT NUMBERS AND '.'
     correct_input  = true;

     input = PromptString("\nEnter current legal US tender amounts as bills/coins one at a time: $");

     //IF STRING IS EMPTY DO NOT PASS
     if (input == "") {
         correct_input = false;
     } else {
         //CHECK FOR INCORRECT CHARS
         foreach (char character in  input ) {
             if (character > 57 || character < 48 && character != 46) {
                 correct_input = false;
             }
         }
     }

     //DISPLAY ERROR
     if(correct_input == false){
         ColorChangeRed   ("\nInput not accepted.");
         Console.WriteLine("Please re-enter legal US tender amounts one at a time...");
     }

 } while (!correct_input);
```

INPUT PAYMENT

```
Enter current legal US tender amounts as bills/coins one at a time: $10

$35.00 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $20

$15.00 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $10

$5.00 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $1

Is this a coin? (y/n)

$4.00 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $
```

Above: User interface

Right: Code snippet from input cash payment function

```
    //PARSE INPUT(STRING) TO DECIMAL
    input_payment = decimal.Parse(input);

    //CHECK IF COIN OR BILL
    if(input_payment == 1) {
        Console.WriteLine("\nIs this a coin? (y/n)");
        coin = Console.ReadKey(true).KeyChar.ToString().ToLower() == "y";
    }
    //LOOP THROUGH ALL US TENDER
    for (int index = 0; index < real_values.Length; index++) {

        //CHECK IF THE INPUT MATCHES LEGAL US TENDER
        if (input_payment == real_values[index]) {
            //ADD DOLLAR COIN TO APPROPRIATE DRAWER
            if (coin) {
                cash_drawer[index+1] += 1;
            } else {
                //ADD BILL TO CASH DRAWER
                cash_drawer[index] += 1;
            }
            //ADD PAYMENT TO TOTAL PAYMENT
            cash_payment += input_payment;
            total_cash_payment += input_payment;

            //GIVE REMAINING COST
            if(cash_payment < total_cost) {
                Console.WriteLine("\n{0:C} Remaining", total_cost - cash_payment);
            }
            //PASS AS CORRECT
            correct_value = true;

            //EXIT LOOP ONCE CORRECT VALUE HAS BEEN FOUND
            break;
        }//end if
    }//end for
    //IF USER DIDN'T ENTER CORRECT VALUE, DISPLAY ERROR.
    if (correct_value == false) {
        ColorChangeRed("\nERROR: Please enter payments one at a time" +
            " as current legal US tender bills/coins.");
        }
}while (cash_payment < total_cost);

return cash_payment;
```

CHECK FOR CHANGE

```
//CALCULATE CHANGE
change = total_payment - total_cost;

//CONVERT CHANGE TO INT & MULTIPLY BY 100 FOR MOD
change_as_int = Convert.ToInt32(change*100);

//TRY TO GET CORRECT CHANGE, ELSE RETURN FALSE
while (change_as_int != 0 && change_possible) {
    change_possible = false;

    for (int index = 0; index < cash_values.Length; index++) {

        //IF THE CURRENT CHANGE AMOUNT CAN BE DIVIDED BY CURRENT COIN, AND WE HAVE ENOUGH CURRENCY IN CURRENT DRAWER
        if ((change_as_int % cash_values[index]) != change_as_int && cash_drawer[index]-(change_as_int/cash_values[index]) >= 0) {

            //GIVES LEFTOVER
            change_as_int %= cash_values[index];

            change_possible = true;

        }//end if
    }//end for
}//end while

//IF UNABLE TO MAKE CHANGE, RETURN FALSE
if (change_as_int > 0) {
    return false;
}
//ELSE RETURN TRUE
return true;
```

OUTPUT CHANGE

```
Enter current legal US tender amounts as bills/coins one at a time: $.50

$11.50 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $10

$1.50 Remaining

Enter current legal US tender amounts as bills/coins one at a time: $2

Change: $0.50

Dispensing $0.50 coin
```

Above: User interface

Right: Code snippet from output change function

```
//GET CHANGE AMOUNT
change = total_payment - total_cost;

//CONVERT CHANGE TO INT FOR MOD
change_to_give = Convert.ToInt32(change*100);

if (cashback){
    ColorChangeYellow($"\nCashback: {change:C}\n");
} else {
    ColorChangeYellow($"\nChange: {change:C}\n");
}

//LOOP TILL ALL CHANGE IS GIVEN
while (change_to_give != 0) {

    for (int index = 0; index < cash_values.Length; index++) {

        //CHECK IF MOD IS POSSIBLE ON CURRENT
        if ((change_to_give % cash_values[index]) != change_to_give &&
            cash_drawer[index]-(change_to_give/cash_values[index]) >= 0){

            // NUMBER OF COINS TO GIVE
            currency_num = change_to_give / cash_values[index];

            //SUBTRACTS TENDER FROM APPROPRIATE BILL/COIN IN CASH_DRAWER
            cash_drawer[index] -= currency_num;

            //DISPLAY GIVEN CHANGE INDIVIDUALLY
            for (int index1 = 0; index1 < currency_num; index1++) {
                ColorChangeGreen($"Dispensing {cash_names[index]}");
            }

            //GET LEFTOVER CHANGE
            change_to_give %= cash_values[index];

        }//end if
    }//end for
}//end while
return change;
```

CONDITIONAL REFUND

```
The total cost of your items is $10.00
Pay with card? (y/n)

Enter current legal US tender amounts as bills/coins one at a time: $20

Please use alternative method of payment. No change available.
Giving refund...

Dispensing $20 bill

Would you like to use another payment method? (y/n)
```

Above: User interface

Right: Code snippet from refund function

Refund calculated by multiplying the total payment by 100, card payment by 100, then subtracting total payment by the card payment.

```
//LOOP TILL REFUNDED
while(refund != 0) {

    //IF ABLE TO MOD REFUND BY THIS CASH VALUE AND THERE IS MONEY IN THAT DRAWER...
    if (refund % cash_values[index] != refund && cash_drawer[index] != 0) {

        //CALC HOW MANY COINS TO REFUND FROM THAT DRAWER
        int temp = (refund / cash_values[index]);

        //DISPLAY REFUNDED BILL/COIN INDIVIDUALLY
        for (int index1 = 0; index1 < temp; index1++) {
            ColorChangeGreen($"Dispensing {cash_names[index]}");
        }

        //MOD REFUND BY CURRENT VALUE
        refund %= cash_values[index];

        //SUBTRACT APPROPRIATE AMOUNT FROM THE APPROPRIATE CASH DRAWER
        cash_drawer[index] -= temp;

    }
    //INCREMENT INDEX
    index++;

}//end while
```

SECOND EDITION – CARD PAYMENT

New addition:
credit card payment

Choice of card or cash payment

```
//ask for card payment
bool card = PromptValidLoop("Pay with card? (y/n)");

//if they pay with card, start card payment loop
if (card) {

    do {
```

Choice of cashback

```
if (validation) {
    //ask for cash-back
    cashback  = PromptValidLoop("\nWould you like cash-back? (y/n)");

    //get cash-back amount
    if (cashback) {
        cashback_amount = PromptDecimal("How much? ");
        //add requested cash-back amount to the remaining cost
        remaining_cost += cashback_amount;
    }
```

BOOLEAN VALIDATION LOOP

```
static bool    PromptValidLoop(string message) {
    bool valid = false;
    string input;
    Console.WriteLine(message);
    do {
        input = Console.ReadKey(true).KeyChar.ToString().ToLower();
        if (input == "y") {
            return true;
        }
        else if(input == "n"){
            return false;
        }
    }
    while(!valid);
    return false;
```

CARD IDENTIFICATION AND VALIDATION

```
//convert string of credit card number to int array
foreach  (char item in CCN) {
    store[count] = Convert.ToInt32(item);
    count++;
}
//reverse the array
Array.Reverse(store);

//start Luhn's formula
//skip drop number (at index 0)
for (int index = 1; index < store.Length; index++) {

    //for every odd positioned number, double the value
    if (index % 2 == 1) {
        store[index] *= 2;

        //if that value is greater than 9, subtract 9
        if (store[index]  > 9) {
            store[index] -= 9;
        }//end nested if
    }//end if

    //add number to total
    total += store[index];

}//end for

//add the drop value to total
total += store[0];

//if mod of total is zero, card is valid
if (total % 10 == 0) {
    return true;
} else {
    return false;
}
```

```
//discover
if (card_id.StartsWith("6")) {
    //take the first two letters and add them to empty string
    for (int index = 0; index < 2; index++) {
        sized += card_id[index];
    }
    if (sized == "64" || sized == "65"){
        Console.WriteLine("DISCOVER"); vendor = "DISCOVER"; return true;
    }
    sized = "";

    //take the first four letters and add them to empty string
    for (int index = 0; index < 4; index++) {
        sized += card_id[index];
    }
    if (sized == "6011"){Console.WriteLine("DISCOVER"); vendor = "DISCOVER"; return true;}
    sized = "";

    //take the first six letters and add them to empty string
    for (int index = 0; index < 6; index++) {
        sized += card_id[index];
    }
    if (int.Parse(sized) >= 622126 && int.Parse(sized) <= 622925
        || int.Parse(sized) >= 624000 && int.Parse(sized) <= 626999
        || int.Parse(sized) >= 628200 && int.Parse(sized) <= 628899){
        Console.WriteLine("DISCOVER"); vendor = "DISCOVER"; return true;
    }
}//end discover
```

Left: Card ID function

Above: Card validation function

SIMULATE REQUEST OF FUNDS FROM CARD ACCOUNT

```
Random rnd = new Random();

//50% CHANCE TRANSACTION PASSES OR FAILS
bool pass     =  rnd.Next(100) < 50;

//50% CHANCE THAT A FAILED TRANSACTION IS DECLINED
bool declined =  rnd.Next(100) < 50;
if(pass) {
    return new string[] { account_number, amount.ToString() };
}else if(!declined) {
    //pays for max of half, min of 1/5
    return new string[] { account_number, (amount / rnd.Next(2,6)).ToString() };
}else {
    return new string[] { account_number, "declined" };
}//end if
```

CALCULATING CORRECT CARD PAYMENT

```
bool     CalculatePayment(string[] card_result, ref decimal remaining_cost, decimal total_cost, ref decimal total_payment, ref decimal cashback_amount, ref decimal total_card_payment) {
```

```
if (card_result[1] != "declined") {
    //store the available funds to current card payment
    current_card_payment = decimal.Parse(card_result[1]);

    //add current payment to the total of card payments
    total_card_payment += current_card_payment;

    if (total_card_payment < remaining_cost && total_card_payment > total_cost) {
        //if the card payment more the cost but less than requested cashback amount, get the difference and subtract it
        decimal difference    = total_card_payment - total_cost;
        total_card_payment   -= current_card_payment;
        current_card_payment -= difference;

        //add the current card payment to the total payment
        total_payment += current_card_payment;
        //remove the cashback amount from the remaining cost
        remaining_cost -= cashback_amount;

        ColorChangeGreen("\nNot enough funds for cashback, base sale approved.");
        Console.WriteLine($"Current card payment: {current_card_payment}");
        Console.WriteLine($"Total paid: {total_payment:C}");
        //reset cashback to zero
        cashback_amount = 0;

        //checks if the paid amount is at least equal to the cost
    } else if (total_card_payment >= total_cost + cashback_amount) {
        // if payment is greater, calculate the difference and subtract it from total payment
        if (total_card_payment > total_cost + cashback_amount) {
            decimal difference = total_card_payment - total_cost - cashback_amount;
            total_card_payment -= difference;
        }
        total_payment += current_card_payment;
        ColorChangeGreen("\nApproved!");
        Console.WriteLine($"Current card payment: {current_card_payment:C}");
        Console.WriteLine($"Total paid: {total_payment:C}");
```

```
    } else {
        //total card payment is less than current cost, aka partial payment

        //remove cashback from the current cost
        remaining_cost -= cashback_amount;
        //add the card payment to the total payment
        total_payment += current_card_payment;
        //cashback_amount = 0.0m;
        if (total_payment < remaining_cost && cashback_amount != 0) {
            //subtract the current payment from card from the current cost
            remaining_cost -= current_card_payment;
            cashback_amount = 0.0m;
            ColorChangeRed("\nIncomplete funds, cannot give cashback.");
            Console.WriteLine($"Current card payment: {current_card_payment:C}");
            Console.WriteLine($"Total paid: {total_card_payment:C}");
            Console.WriteLine($"Remaining balance: {remaining_cost:C}.");
        } else {
            //subtract the current payment from card from the current cost
            remaining_cost -= current_card_payment;
            cashback_amount = 0.0m;
            ColorChangeRed("\nIncomplete funds.");
            Console.WriteLine($"Current card payment: {current_card_payment:C}");
            Console.WriteLine($"Total paid: {total_card_payment:C}");
            Console.WriteLine($"Remaining balance: {remaining_cost:C}.");
        }
    }
return false;

} else {
    return true;
}
```

REFUND FUNCTION MODIFIED IN CASE OF CASHBACK FAILURE

```
//MULTIPLY TOTAL PAYMENT BY 100 FOR MOD, STORE AS INT
int refund = Convert.ToInt32(total_payment*100 - card_payment*100);

if (cashback) {
    Console.WriteLine($"\nNot enough change available to give cashback." +
        $"\n{cashback_amount:C} will not be charged.");
    //remove cashback amount from the card payment
    card_payment -= cashback_amount;
} else {
    ColorChangeRed   ("\nPlease use alternative method of payment. No change available.");
    Console.WriteLine("Giving refund...\n");
```

THIRD EDITION – LOGGING SALES

- Date and time
- Transaction count, if sale was valid
- Cash payment
- Card payment
- Card vendor
- Change given

CONVERT GIVEN VARIABLES TO STRING AND FEED TO NEW PROGRAM

```
static string   LogArguments    (int transaction_number, string transaction_time, decimal total_cash_payment, decimal total_card_payment, string vendor, decimal change) {
    //adds each variable to a string, insert a space between each one.
    string result = "";
    result += transaction_number.ToString() + " " +
        transaction_time + " " +
        "$"+total_cash_payment.ToString("0.00") + " " +
        "$"+total_card_payment.ToString("0.00") + " " +
        vendor + " " +
        "$"+change.ToString("0.00");
    return result;
```

```
//if valid sale, append sales information to log
if (!canceled) {
    result = LogArguments(transaction_number, transaction_time, total_cash_payment, total_card_payment, vendor, change);

    //start new process
    ProcessStartInfo startInfo = new ProcessStartInfo();
    //point to logging program
    startInfo.FileName  = "Changebot Transaction Log.exe";
    //feed program info as string, vars separated by spaces
    startInfo.Arguments = result;
    //run program
    Process.Start(startInfo);
}
```

ADDITIONAL PROGRAM – TRANSACTION LOG

```
static void Main(string[] args) {
    //get current date, convert and save to string
    DateTime dt = DateTime.Now;
    string filename = dt.ToString("MM-dd-yyyy");

    string[] clarifiers = {"Transaction:\t", "Date:\t\t", "Time:\t\t", "Cash Payment:\t", "Card Payment:\t", "Vendor:\t\t", "Change:\t\t"};

    WriteToFile(args, clarifiers, filename);
}

1 reference
static void WriteToFile(string[] args, string[] clarifiers, string file_name) {
    StreamWriter out_file;
    out_file = new StreamWriter(file_name + ".log",true);

    //writes each item from the given array
    for (int index = 0; index < args.Length; index++) {
        out_file.Write(clarifiers[index]);
        out_file.WriteLine(args[index]);
    }
    //double new line for readability
    out_file.WriteLine("\n");
    out_file.Close();
}//end writetofile()
```

CHALLENGES

- Poor planning
  - Did not consider all the possible situations a customer could find themselves in, leading to many bugs that seem trivial in hindsight
- Coding on the fly
  - Spent too much time on certain areas, causing a rush to complete the project towards the end, which in turn created new bugs
- Overpayment
  - Random nature of the MoneyRequest function showed us flaws in our logic