Chaincode is programmed using the *Go* language and framework.

You will not need to learn how to code to complete this mission, however you are welcome to browse the basics below.

GO BASICS

```
package main

import (
	//"bytes"
	"encoding/json"
	"fmt"
	"strconv"
	//"strings"
	"time"

	"github.com/hyperledger/fabric/core/chaincode/shim"
	pb "github.com/hyperledger/fabric/protos/peer"
)

//==============================================================================================
==============================================================================================

//
                                                Structs

//==============================================================================
==============================



type SimpleChaincode struct {
}


type PurchaseOrder struct {

	PONumber 		string `json:"ponumber"`
	ItemNumber 		string `json:"itemnumber"`
	ItemDescription string `json:"itemdescription"`
	Quantity 		string `json:"quantity"`
	PricePerUnit	string `json:"priceperunit"`
	ExtendedPrice	string `json:"extendedprice"`
	ShippingTerms	string `json:"shippingterms"`
	OrderStatus 		string `json:"orderstatus"`
	PurchaseDate	string `json:"purchasedate"`



	// add quality
}
```

TRUEUP TIP

A struct is a collection of fields

Think about the fields you might include in a source document, such as an invoice, sales order, vendor payment, inventory transfer, etc.

```
//
============================================================================================
============================================================

//
                                                MAIN FUNCTIONS

//
============================================================================================
============================================================
func main() {
        err := shim.Start(new(SimpleChaincode))
        if err != nil {
                fmt.Printf("Error starting Simple chaincode: %s", err)
        }
}

// Init initializes chaincode
// ===========================
func (t *SimpleChaincode) Init(stub shim.ChaincodeStubInterface) pb.Response {
        var err error
        err = stub.PutState("Smartify Purchase Contract", []byte("BLOCKED!  TrueUp did not accept the terms in
your chaincode. Something is wrong!")) //write the variable into the chaincode state
        if err != nil {
                return shim.Error(err.Error())
        }

        return shim.Success(nil)
}

// Invoke - Our entry point for Invocations
// ========================================
func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface) pb.Response {
        function, args := stub.GetFunctionAndParameters()
        fmt.Println("invoke is running " + function)

        // Handle different functions
        if function == "makePurchaseOrder" { //create a new user
                return t.approvePurchaseOrder(stub, args)
        }  else if function == "read" {
                return t.read(stub, args)
        }
        fmt.Println("invoke did not find func: " + function) //error
        return shim.Error("Received unknown function invocation")
}
```

TRUEUP TIP

A function can take zero or more arguments.

Some accounting functions may include creating, updating, approving, or reading a transaction

```
//============================================================================================
============================================================================================

//
                                        REGISTRATION CODE BELOW

//============================================================================================
============================================================================================

func (t *SimpleChaincode) makePurchaseOrder(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var err error

//          PONumber                string `json:"ponumber"`
//          ItemNumber              string `json:"itemnumber"`
//          ItemDescription string `json:"itemdescription"`
//          Quantity                string `json:"quantity"`
//          PricePerUnit     string `json:"priceperunit"`
//          ExtendedPrice   string `json:"extendedprice"`
//          PaymentTerms  string `json:"paymentterms"`
//          ShippingTerms  string `json:"shippingterms"`
//          PurchaseDate    string `json:"purchasedate"`



        // ==== Input sanitation ====

        ponumber := args[0]
        itemnumber := args[1]
        itemdescription := args[2]
        quantity := args[3]
        unitprice := args[4]

        i, err := strconv.Atoi(quantity)
        i2, err := strconv.Atoi(priceperunit)
                if err != nil {
      // handle error
      fmt.Println(err)
      }

        extendedprice := strconv.Itoa(i*)


        paymentterms := args[5]
        shippingterms := args[6]
        orderstatus := args[7]
        var time = time.Now()
        var formatTime = time.Format("20060102150405")
        purchasedate := formatTime
```

TRUEUP TIP

This section called "Registration Code" represents code to generate the new purchase order and record it in the block.

Note, anytime you see a stream of equals signs, such as "=====", it is ***not*** functional code, rather it is an engineer's way to add "notes to self" within a coded file.

```
	// ==== Check if RM already exists ====
	rawidAsBytes, err := stub.GetState(ponumber)
	if err != nil {
		return shim.Error("Failed to get user: " + err.Error())
	} else if rawidAsBytes != nil {
		fmt.Println("This user already exists: " + ponumber)
		return shim.Error("This user already exists: " + ponumber)
	}

	// ==== Create RM object and marshal to JSON ====

	PurchaseOrder := &PurchaseOrder{ponumber, itemnumber, itemdescription, quantity, priceperunit,
	paymentterms, shippingterms, orderstatus, purchasedate}
	PurchaseOrderJSONasBytes, err := json.Marshal(PurchaseOrder)
	if err != nil {
		return shim.Error(err.Error())
	}


	// === Save RM to state ===
	err = stub.PutState(ponumber, PurchaseOrderJSONasBytes)
	if err != nil {
		return shim.Error(err.Error())
	}

	//  Index
	indexName := "ponumber~itemnumber"
	rawidIndexKey, err := stub.CreateCompositeKey(indexName, []string{PurchaseOrder.PONumber,
PurchaseOrder.ItemNumber})
	if err != nil {
		return shim.Error(err.Error())
	}
	//  Save index entry to state. Only the key name is needed, no need to store a duplicate copy of the user.
	//  Note - passing a 'nil' value will effectively delete the key from state, therefore we pass null character as
value
	value := []byte{0x00}
	stub.PutState(rawidIndexKey, value)

	// ==== RM saved and indexed. Return success ====
	fmt.Println("- end init user")
	return shim.Success(nil)
}


//
```

```
================================================================================
================================================================================

//
                                    Reading

//==============================================================================
================================================================================


func (t *SimpleChaincode) read(stub shim.ChaincodeStubInterface, args []string) pb.Response {
        var A string // Entities
        var err error

        A = args[0]

        // Get the state from the ledger
        Avalbytes, err := stub.GetState(A)
        if err != nil {
                jsonResp := "{\"Error\":\"Failed to get state for " + A + "\"}"
                return shim.Error(jsonResp)
        }



        //jsonResp := "{\"Name\":\"" + A + "\",\"Amount\":\"" + string(Avalbytes) + "\"}"
        //fmt.Printf("Query Response:%s\n", jsonResp)
        return shim.Success(Avalbytes)
}
```

TRUEUP TIP

The code within this section allows someone (i.e. accountant, auditor, data analyst, etc.) on the front end of their system to query and READ records from the block ledger.