Assignment 2

1) Here, the below code shows the bond present value calculation by discounting future values.

```
<!DOCTYPE html>
<html>
<head>
	<title></title>
</head>
<body>
	<script type="text/javascript">


		// Declared Variables for User Input
var c = prompt("Enter Annual Coupon Rate %:", "");
var cf = prompt("Enter Coupon Payment Frequency:")
var i = prompt("Enter Yield %:", "");
var fv = prompt("Enter Face Value:", "");
var n = prompt("Enter Maturity Date", "");

//Bond Price Calculation Function
function bondPrice(c, cf, fv, n) {
  var r = (i / 100) / cf
  var cpmnt = ((c / 100) / cf) * fv
  var bondprice = (cpmnt*((1-(Math.pow((1+r),-n*cf)))/r)) + (fv*(Math.pow((1+r),-n*cf)))

console.log(bondprice)
}

bondPrice(c, cf, fv, n)
	</script>

</body>
</html>
```

Assignment 2

2) This codes shows only the condition required to fulfill the buying / selling function for bonds and transferring the value to seller from the buyers account.

```
pragma solidity ^0.5.1;



// Bond Smart contract
pragma solidity >=0.4.22 <0.6.0;

contract Bond{
   uint public number_of_bonds = 1;

   uint public price_of_bond = 100;

   uint public intial_position = 0;

   //mapping of investors address
   // mapping is an array
   mapping(address => uint) equity_bond;

   mapping(address => uint) equity_usd;

   // Check if can buy the Bond
   modifier can_buy_bond(uint usd_invested){
      require (usd_invested * number_of_bonds + intial_position <= price_of_bond);
      _; // The _ is say that the functions we will link the modifier with only be applied if
condition met
   }

   //Get if the investor owns the Bond. external constant: never modified and not proper to
contract
   function equity_in_bond(address bondholder) external view returns(uint){
      return equity_bond[bondholder];
   }
```

```
    // Get if the investor has the USD
    function equity_in_usd(address bondholder) external view returns(uint){
        return equity_usd[bondholder];
    }

    // Buy Bond and apply modifier
    function buy_bond(address investor, uint usd_invested) external
    can_buy_bond(usd_invested){
        uint bond_bought = usd_invested * number_of_bonds;
        equity_bond[investor] += bond_bought;
        equity_usd[investor] = equity_bond[investor] * price_of_bond;
        intial_position += bond_bought;
    }
    // Sell bond
    function sell_bond(address investor, uint bond_sold) external{
        equity_bond[investor] -= bond_sold;
        equity_usd[investor] = equity_bond[investor] * price_of_bond;
        intial_position -= bond_sold;
    }
}


contract PurchaseBond {
    uint public valuebond;
    address public seller;
    address public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() public payable {
        seller = msg.sender;
        valuebond = msg.value / 2;
        require((2 * valuebond) == msg.value, "Value has to be even.");
    }

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
```

```
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
```

Assignment 2

```
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * valuebond))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        // NOTE: This actually allows both the buyer and the seller to
        // block the refund - the withdraw pattern should be used.

        buyer.transfer(valuebond);
        seller.transfer(address(this).balance);
    }
}
```