CPSC-354 Report

Tyler Lewis
Chapman University

January 17, 2023

Abstract

Section 1: Project This report details a research project and homework assignments per Chapman Univ. CPSC 354: Programming Languages. Within this section of the document is a detail on the Solidity programming language within the Ethereum Virtual Machine (EVM): including Solidity's programming semantics, security considerations, and includes learn-by-example exercises, with a subsequent programming project exploring the contracting of assets within a Solidity blockchain ecosystem.

Section 2: Homework is of varying topics.

Contents

1 Project

1.1 Introduction

1.1.1 Abstract

In recent years, the use of blockchain technology has gained significant momentum in a variety of industries. One of the key features of blockchain technology is the ability to use smart contracts to create and manage digital assets and tokens. Solidity is a popular programming language used for developing smart contracts on the Ethereum blockchain. This research project aims to explore how assets and tokens are handled within the Solidity language and evaluate its capabilities and limitations in managing digital assets and tokens.

In addition to exploring the handling of assets and tokens in Solidity, this research project will also provide an introduction to the fundamentals of Solidity and blockchain programming. This will include an overview of the key concepts and principles of Solidity, as well as a discussion of the unique challenges and opportunities that come with programming on the blockchain. Through this report, readers will gain a solid foundation in Solidity and blockchain programming, and be enabled to better understand and evaluate the handling of assets and tokens within the Ethereum Virtual Machine.

1.1.2 Foundational knowledge

Blockchain is a distributed ledger technology that enables a network of computers to maintain a shared, immutable record of transactions. Each transaction is recorded in a block, which is cryptographically linked to the previous block in the chain. This creates a permanent and tamper-evident record of all transactions on the network.

Blockchain programming involves developing applications that run on blockchain platforms and make use of their unique features, such as decentralization and immutability. These applications are often called dapps (decentralized applications) and can include a wide range of use cases, from financial services and supply chain management to internet fad economys and enabling pseudo-anonymous transactions.

Solidity is a programming language used for writing smart contracts on blockchain platforms. It is designed to be backwards-compatible with the Ethereum Virtual Machine (EVM). Solidity is a statically-typed, object-oriented programming language. It is influenced by C++, Python, and JavaScript, and is designed to be familiar to developers with experience in those languages.

Smart Contracts are what Solidity programs are called, *similar to a class in Java*, and serve as self-executing pieces of code that run on the EVM. Smart contracts can interact with the blockchain and with other smart contracts, allowing them to store and transfer data and assets on the network.

Ethereum Virtual Machine (EVM) The runtime environment for smart contracts on the Ethereum blockchain.

1.1.3 Safety and Security

Solidity also has a number of safety and security features that are important for blockchain programming.

Type System One of Solidity's key security features is its strict type system. In Solidity, variables and function arguments must be explicitly declared with their type, and the type of a variable cannot be changed once it has been declared. This helps prevent type-related vulnerabilities, such as type confusion attacks, in which a malicious actor is able to manipulate the type of a variable and cause the contract to behave in unexpected ways.

Access Control Solidity also includes support for access control, which allows developers to specify which addresses are allowed to call which functions in a contract. This can be used to restrict the actions that can be performed on a contract, and to prevent unauthorized access to sensitive data or functions. Access control is implemented using modifiers 1.2.5, which can be used to define a set of conditions that must be met in order for a function to be executed. For example, a `modifier` might be used to specify that only the contract owner is allowed to call a particular function.

1.1.4 Ethereum development

Open source development for Ethereum happens through a decentralized process that involves a wide range of participants, including developers and users in the Ethereum community.

Any member of the community can propose a change or improvement to the Ethereum platform through an Ethereum Improvement Proposal [EIP]. Such improvements include application contract standards and conventions such as for tokens 1.4, interface standards 1.3.3, and additionally for core Ethereum functionality.

1. The EIP is then reviewed by the community, which will provide feedback and suggestions for improvement.
2. If accepted by the community, it is assigned an EIP number and becomes an official EIP.
3. The EIP is then implemented in the Ethereum codebase and tested to ensure it works as intended.
4. If the EIP is successfully implemented and tested, it can be adopted as an `Ethereum Request for Comment` (ERC) and become a part of the Ethereum protocol.

This process is designed to ensure that changes to the Ethereum platform are transparent, decentralized, and subject to review by the community. It allows for a wide range of perspectives to be considered and for the Ethereum platform to evolve and improve over time.

EIP20 and EIP721 are discussed in detail in section 1.4

1.2 How to speak Solidity

Solidity is a programming language for writing smart contracts Solidity has several important features that make it well-suited for blockchain and smart-contract programming.
These include:

- Support for defining complex data structures, such as arrays, enums, and structs 1.2.3
- Support for defining and implementing custom functions and logic 1.2.2
- Support for inheritance and polymorphism, allowing for the creation of complex, modular code 1.3
- Built-in support for common blockchain operations, such as storing and transferring assets 1.4

1.2.1 Basic Solidity rules and features

- **License identifier** at top of .sol file
- **Solidity** version identifier
- **Contract** declared with name
- **State variables** declared top of contract
- **Constructor**, with parameters, and only required for most child contract 1.3.1

- **Functions**, with parameters
- **Require statements**, no execution unless statement evaluates to true
- **msg** global variable is a struct containing information regarding a contract interaction

1.2.2 Writing a simple smart contract

Writing a Solidity smart contract involves defining the contract's data structures, functions, and logic, and then deploying the contract to the EVM.

To define a smart contract in Solidity, a developer must first create a new file with a .sol extension. This file can then be used to define the contract's data structures and functions.

Before we discuss basics, we will explore a simple smart contract called AuctionTracker that tracks a highest bidder in an auction and requires a minimum increment. The contract uses each of the aspects listed in section 1.2.1, try to identify them all in the code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

contract AuctionTracker {

    uint16 public bidincrement;
    address public owner;

    address public highestbidder;
    uint256 public highestbid;

    constructor(uint16 _bidincrement){
        owner = msg.sender;
        highestbid = 0;
        bidincrement = _bidincrement;
    }

    function bid(uint256 _value) public {
        require(_value >= highestbid + bidincrement, "bid minumum not met");
        highestbidder = msg.sender;
        highestbid = _value;
    }
}
```

1.2.3 Data types

There are several types of variables that can be used to store different types of data.
These include:

- **bool**: a type that represents a boolean value, either true or false
- **int**: a signed integer type that can store values from -(2^{256}) to $2^{256\text{-}1}$
- **uint**: an unsigned integer type that can store values from 0 to 2^{256}
- **address**: a type that represents an Ethereum address, which is a 20-byte value
- **bytes**: a dynamic-sized byte array
- **string**: a dynamic-sized string of Unicode characters

Arrays, structs, and special types:

- **array**: a type that represents an ordered sequence of elements
- **struct**: a type that represents a composite data structure, consisting of a collection of named members
- **mapping**: a type that represents a collection of key-value pairs, where the keys are unique and the values can be any type
- **enum**: a type that represents a set of named values. Refer to 1.6.3 for usage in contract

More types These are just some of the types available in Solidity, variable type int and uint, for instance, are defaulted to 256-bits, but can me adjusted for specific bit-sizes. For a complete list and more detailed information about the different types, you can refer to the [Solidity Language Documentation].

1.2.4 Global variables

`msg`, `this`, and `block` are all examples of global object variables in Solidity that are available to all functions within a contract.

- `msg` represents the current message being processed. It contains information about the sender of the message, the function that is being called, and any arguments passed to the function.
- `this` represents the current contract instance. It can be used to access the contract's functions and variables from within the contract.
- `block` represents the current block being processed. It contains information about the block's number, timestamp, and other properties.

It's important to note that these global variables are read-only and cannot be modified. They are intended to provide information about the current context. Usage of global variables can be easily inferred through usage in further examples in this report or looking at the global variable properties documentation.

1.2.5 Modifiers

Solidity provides several built-in modifiers that you can use to alter the behavior of functions and variables in your contract. Here is a list of some of the most commonly used built-in modifiers in Solidity:

- **public**: makes a function or variable accessible to any contract or external caller.
- **private**: makes a function or variable only accessible to functions within the same contract, and not accessible to external callers or other contracts.
- **internal**: makes a function or variable only accessible to other functions within the same contract.
- **view**: indicates that a function only reads data from the blockchain and does not modify it. This allows the function to be called without the need to create a transaction.
- **pure**: indicates that a function does not read or modify the state of the contract or the blockchain, and does not have any side effects.
- **payable**: indicates that a function can receive Ether (the native cryptocurrency of the Ethereum blockchain) as an input.
- **memory**: indicates that a function or variable should be stored in memory, rather than in storage.
- **storage**: indicates that a function or variable should be stored in storage, rather than in memory.

- **virtual**: indicates that a function can be overridden by a derived contract

You can use these built-in modifiers in combination with function and variable definitions to control their behavior. For example, you might define a function as `public payable` to make it both accessible to external callers and able to receive Ether as an input.

Custom modifiers To create a custom modifier in Solidity, you can use the modifier keyword followed by the name of the modifier and a block of code that defines its behavior. The modifier implementation must end with `_;`

Here's an example of a custom modifier that checks if the caller of a function is contained in `address owner`:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

contract Example {

   public address owner;
   public uint price;

   modifier onlyOwner() {
      require(msg.sender == owner, "Only the owner can perform this action");
      _;
   }

   function changePrice(uint _price) onlyOwner {
      price = _price;
   }
}
```

The function `changePrice()` is only accessible when the modifier `onlyOwner()` is satisfied. Modifier can also accept arguments.

1.3 Inheritance and Polymorphism

1.3.1 Inheritance

Inheritance is a mechanism that allows a contract (called the `ChildContract`) to inherit the state and behavior of another contract (called the `ParentContract`). This means that the child contract automatically has all the state variables and functions defined in the parent contract, and it can also define its own state variables and functions.

Inheritance is useful because it allows contracts to share common code, which can make the code easier to understand and maintain. It also allows contracts to be easily extended or modified by creating a new contract that inherits from an existing contract and adding new functionality to it.

To create a contract that inherits from another contract in Solidity, the `is` keyword is used followed by the name of the parent contract. For example:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

// Define modular contract functions and variables
contract ParentContract { /*...state variables, functions */ }
```

```
// Inherits ParentContract functions and variables
contract ChildContract is ParentContract { /*...state variables, constructor, functions */ }
```

In this example, the `ChildContract` contract inherits all of the properties and functionality of the `ParentContract` contract. This means that the `ChildContract` contract can access any public or internal functions and state variables defined in `ParentContract`.

Additionally, the `ChildContract` contract can override any functions from the `ParentContract` contract by defining a function with the same name and signature. This allows the `ChildContract` contract to customize the behavior of inherited functions.

Inheritance is a powerful feature of Solidity that allows developers to create flexible, modular contracts. It is an important concept to understand when working with Solidity and blockchain development.

1.3.2 Polymorphism

Polymorphism in Solidity refers to the ability of a contract or function to behave differently based on the type or number of arguments passed to it. This can be achieved through the use of function overloading, in which multiple functions with the same name can be defined, but with different amounts or types of arguments.

Consider the following:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

contract PolymorphismExample {
  function multiply(uint x, uint y) public pure returns (uint) {
    return x * y;
  }

  function multiply(uint x, uint y, uint z) public pure returns (uint) {
    return x * y * z;
  }

  function multiply(uint[] values) public pure returns (uint) {
    uint result = 1;
    for (uint i = 0; i < values.length; i++) {
      result *= values[i];
    }
    return result;
  }
}
```

When calling the multiply function, the executed function will be determined based on the number and type of arguments provided. For example, if we call `multiply(2, 3)`, the first version of the function will be executed and will return 6. If we call `multiply([2, 3, 4])`, the third version of the function will be executed and will return 24. This allows for greater flexibility and reusability of the multiply function.

The actual function that is executed will be determined at runtime based on the arguments provided. Polymorphism allows for more flexible and reusable code, as the same function can be used in different ways depending on the context. However, it can also make the code more complex and difficult to read and understand.

1.3.3 Interfaces

Polymorphism can also be achieved across multiple contracts through the inheritance of interfaces. An interface is a contract that defines a set of functions that other contracts can implement. This allows for polymorphic behavior, as multiple contracts can implement the same interface and provide their own unique implementations of the functions defined in the interface.

Here is an example of how this might work:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

// Define an interface that specifies a function signature
interface Shape {
  function getArea() public view returns (uint);
}

// Define a contract that implements the Shape interface
// and provides its own implementation of the getArea function
contract Circle is Shape {
  uint radius;

  function Circle(uint r) public {
    radius = r;
  }

  function getArea() public view returns (uint) {
    return radius ** 2 * 3.1415;
  }
}

// Define another contract that also implements the Shape interface
// and provides its own implementation of the getArea function
contract Rectangle is Shape {
  uint width;
  uint height;

  function Rectangle(uint w, uint h) public {
    width = w;
    height = h;
  }

  function getArea() public view returns (uint) {
    return width * height;
  }
}
```

`Circle` and `Rectangle` are both `Shape` extensions, with a varying implementation of function `getArea()`

To examine production level interface contracts and more, take a look at the [Open Zeppelin's solidity library] For OpenZeppelin contracts pertaining to subject in this report, navigate to the `/contracts/token/ERC721` directory within the repository.

Interfaces in OpenZepplin are identified by a file name with a prefix of `I`, as in: `I[interface-name].sol`

1.4 Assets

Begin by understanding the concepts of EIP and ERC from Section 1.1.4

1.4.1 Fungible tokens

Fungible tokens are blockchain assets which can be exchanged for another unit of the same token without any loss of value. An example of a fungible token is a digital currency, such as Bitcoin or Ethereum, where each individual token is worth the same amount and can be easily exchanged for another token of the same type. Similarly, a governance token for a DAO (Decentralized Organization) would likely be a fungible token where voting power is determined by quantity of tokens, rather than a particular token.

ERC-20 is the most widely used token standard on the Ethereum network, and is a verified contract interface designed to facilitate a transferable and interchangeable token.

1.4.2 Non-fungible tokens

Non-fungible tokens, or NFTs, are blockchain assets which are typically unique and differentiable from one and another. Each non-fungible token is distinct and cannot be replaced by another token. NFTs have gained popularity in the field of digital ownership. Currently, most digital non-fungible ownership is in regard to art and digital collectibles, but in the future may be recognized for a wider array of ownership such as property liens, vehicle titles, and other physical assets.

ERC-721 is a token standard for creating non-fungible tokens on the Ethereum network. The standard specifies function names and variables that every ERC721 token contract possesses. Above anything else, each token associates with an identifier, which inherently makes each token unique.

1.5 Analyzing ERC721

Lets begin by observing the OpenZeppelin ERC721 interface found in the [Open Zeppelin's solidity library] to see how assets handle being transferred. This contract will serve as a guide to learning how to support ERC721 features in a contract of our own.

1.5.1 approve() and setApprovalForAll()

The functions `approve()` and `setApprovalForAll()` are important aspects of ERC721 and allow a token owner to delagate control of their tokens to another entity.

Lets take a look at these functions implementations in the `IERC721 interface` mentioned above 1.5:

```
/**
 * @dev Gives permission to ‘to‘ to transfer ‘tokenId‘ token to another account.
 * The approval is cleared when the token is transferred.
 *
 * Only a single account can be approved at a time, so approving the zero address clears
     previous approvals.
 *
 * Requirements:
 *
 * - The caller must own the token or be an approved operator.
 * - ‘tokenId‘ must exist.
 *
 * Emits an {Approval} event.
 */
function approve(address to, uint256 tokenId) external;

/**
 * @dev Approve or remove ‘operator‘ as an operator for the caller.
 * Operators can call {transferFrom} or {safeTransferFrom} for any token owned by the caller.
 *
 * Requirements:
 *
 * - The ‘operator‘ cannot be the caller.
 *
 * Emits an {ApprovalForAll} event.
 */
function setApprovalForAll(address operator, bool _approved) external;
```

Remember that this is an interface [1.3.3] and contains only the function declaration, rather than including implementation.

The approve() function allows an owner of an ERC721 token to give another address (called the ”approved address”) the ability to transfer the token on behalf of the owner. This is similar to the approve function in the ERC20 token standard, which allows an owner to give another address the ability to transfer ERC20 tokens on behalf of the owner.

The setApprovalForAll() function allows an owner of an ERC721 token to set or unset an approval flag for all of their tokens for a given asset type, or simply NFT assets of the same contract. When the flag is set to true, it means that the owner has given the approved address the ability to transfer all of their ERC721 tokens on their behalf. When the flag is set to false, it means that the approved address no longer has the ability to transfer the tokens on behalf of the owner.

Both the `approve` and `setApprovalForAll` functions can be useful in situations where an owner wants to delegate the ability to transfer their ERC721 tokens to another address, such as a marketplace or a smart contract. However, it’s important to note that these functions should be used with caution, as they allow another address to transfer the tokens on behalf of the owner, which can potentially result in the loss of control over the tokens.

1.5.2 safeTransferFrom()

According to the `IERC721 interface` from which we are currently observing 1.5, an ERC721 token implements the following function safeTransferFrom() to perform transfer of ownership of the token from the current owner to to a suitable Ethereum address:

```
/**
 * @dev Safely transfers ‘tokenId‘ token from ‘from‘ to ‘to‘, checking first that contract
    recipients are aware of the ERC721 protocol to prevent tokens from being forever locked.
 *
 * Requirements:
 *
 * - ‘from‘ cannot be the zero address.
 * - ‘to‘ cannot be the zero address.
 * - ‘tokenId‘ token must exist and be owned by ‘from‘.
 * - If the caller is not ‘from‘, it must have been allowed to move this token by either
    {approve} or {setApprovalForAll}.
 * - If ‘to‘ refers to a smart contract, it must implement
    {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
 *
 * Emits a {Transfer} event.
 */
function safeTransferFrom(
    address from,
    address to,
    uint256 tokenId
) external;
```

Remember that this is an interface [1.3.3] and contains only the function declaration, rather than including implementation.

The documentation in the snippet above indicates that the ERC721 function, `safeTransferFrom`, checks the ability of the receiving Ethereum address to accept ERC721 tokens before initiating a transfer by using onERC721Recieved 1.5.3. **This is the method which a receiving contract must implement to indicate it is able to receive ERC721 tokens**. The purpose of this is to prevent tokens from being sent to a recipient who is unable to interact with the tokens, which would cause the tokens becoming inaccessible or lost. The function always returns the onERC721Received.selector object which is checked by the NFT contract using a require and operator statement nested in the implementation of `safeTransferFrom` making sure that the `onERC721Received()` method exists and returns the proper identifier.

1.5.3 onERC721Received()

The following contract, `ERC721Holder`, contains an implementation of the `onERC721Received` function required by safeTransferFrom() 1.5.2.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.8;

contract ERC721Holder {
    /** Always returns `IERC721Receiver.onERC721Received.selector`. */
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) pure public override returns (bytes4)
    {
        return this.onERC721Received.selector;
    }
}
```

A child contract will become able to receive NFTs by inherriting `ERC721Holder`. In the next section we will work on extending this contract.

1.6 Developing a decentralized application

Lets explore what we have learned about assets transacted by EVM smart contracts with a relatively simple, but foundational smart-contract application.

As with any software development, we begin by identifying the purpose of our application. For this example, and in order to exemplify particular details including solidity asset-management security and contract inheritance, we will build a ERC721 auction application, managed by the EVM.

1.6.1 Project description

For the conclusion of this report, we will combine many of the discussed ideas into a project. I am interested in building an intermediary contract for transacting blockchain assets.

1.6.2 Extending 1.5.3 ERC721Holder contract

Currently, our NFT holder contract from section 1.5.3 performs one purpose: to indicate that it is acceptant of ERC721, however we have nothing in the contract that allows us to give any functionality to managing and interacting with an asset held by the contract.

The contract must be modified to provide functions for depositing and withdrawing an asset from the contract, along with any accessors for the NFT data.

Our first step will be to define state variables (1.2.3) and function modifiers (1.2.5) following what we now know of the Solidity programming language:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

contract ERC721Holder {
    /** These are state variables and must be kept track of carefully to prevent token loss */
    address public nftAddress;
    uint256 public tokenID;

    /** The hasToken() modifier will ensure an asset is present in the contract */
    modifier hasToken() {
        require(tokenID != 0, "There is no asset held in the contract");
        _;
    }
    /** The noToken() modifier will ensure that no token is present in the contract before
     * the function is executed */
    modifier noToken() {
        require(tokenID == 0, "There is already an asset in the contract");
        _;
    }

    /** ...contract functions... */

    /** Always returns `IERC721Receiver.onERC721Received.selector`. */
    function onERC721Received(
        address operator,
        address from,
        uint256 tokenId,
        bytes calldata data
    ) pure public override returns (bytes4)
    {
        return this.onERC721Received.selector;
    }
}
```

Take a moment to examine what we have just added to the contract

1. State variables for keeping track of NFT address and ID number:

- `address public nftAddress` will be used to store the address of an ERC721 token contract.
- `uint256 public tokenID` stores the unique ID of an ERC721 token.

2. Modifier functions [1.2.5] to check if the contract contains a token or not:

- `hasToken()` ensures that a token is present in the contract before a function is executed.
- `noToken()` ensures that no token is present in the contract before a function is executed.

Lets use these variables and modifiers to build functions that implement deposit, withdraw, and can access asset metadata:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract ERC721Holder {

    /** ...variables, modifiers, onERC721Received()... */

    /** noToken, public, virtual function: overload in child to add further functionality */
    function depositAsset(
        address _NFTAddress,
        uint256 _TokenID
    ) noToken public virtual
    {
        require( // check token approval for contract address
            ERC721(_NFTAddress).isApprovedForAll(msg.sender, address(this)),
            "NFT is not approved for transfer"
        );
        /** Create temp ERC721 instance to perform a function on the token */
        ERC721(_NFTAddress).safeTransferFrom(msg.sender, address(this), _TokenID);
        /** Transfer successful if got to this point, save data */
        nftAddress = _NFTAddress;
        tokenID = _TokenID;
    }

    /** hasToken, internal: only visible to functions within the contract system.
     * Performs no owner-check control. Must be done in child. */
    function withdrawAsset(
        address _receiver
    ) hasToken internal
    {
        ERC721(nftAddress).safeTransferFrom(address(this), _receiver, tokenID);
        nftAddress = address(0); // clear nft data by returning to zero address
        tokenID = 0;
    }

    /** hasToken, public, view: This function costs zero to execute, as it does not change
     * the contract state, simply returns a string from the blockchain */
    function getTokenURI()
    hasToken public view
    returns (string memory)
    {
        string memory _return = ERC721(nftAddress).tokenURI(tokenID);
        return _return;
    }
}
```

Once again, lets break down what we have added:

1. Import statement

- import "@openzeppelin/.../ERC721.sol"; allows our contract to have access to the ERC721 standard library within [Open Zeppelin's solidity library]. This is particularly useful because although our contract itself is not an NFT, it needs to understand how to interact with one.

2. Functions

- depositAsset() takes two arguments: an address representing the address of an ERC721 token contract, and a uint256 representing the ID of an ERC721 token. The `noToken` modifier is applied to the function to ensure that no token is present in the contract before the function is executed. The `virtual` keyword indicates that this function can be overridden by a derived contract.

- withdrawAsset() takes one argument: an address representing the address of the recipient of the ERC721 token. The `hasToken` modifier is applied to the function to ensure that a token is present in the contract before the function is executed. The `internal` keyword indicates that the function is only visible to functions within the contract system.

- getTokenURI() applies the `hasToken` modifier to ensure that a token is present in the contract before the function is executed. The `public` keyword indicates that the function can be called by anyone, and the `view` keyword indicates that the function does not modify the state of the contract and only returns a value. The function returns a string representing the URI/metadata of the ERC721 token.

1.6.3 Building a child contract

Lets build a child contract to contract ERC721Holder we built in 1.6.2

The contract we are about to build is defined as contract NFTListing and will contain functions to manage an NFT sale listing with an option to limit the sale for a specific buyer. I am hoping that this report will serve as a suitable guide for understanding further code, which I will detail less from this point on, but I have also included more in-depth comments within the actual code.

Constructing variables (1.2.3) and modifiers (1.2.5) for contract: NFTListing

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

/** ERC721Holder contract provides the basic functionality for custodianship of ERC721 tokens
    */
contract NFTListing is ERC721Holder {
    /** ProjectState enum defines the possible states of the contract */
    enum ProjectState {emptyVault, fullVault, onSale}
    /** projectState variable stores the current state of the contract */
    ProjectState public projectState;
    /** owner variable stores the address of the owner */
    address payable public owner;
    /** buyer variable stores the address of the current buyer (if any) */
    address public buyer;
    /** price variable stores the price of the NFT being sold */
    uint256 public price;

    /** inState modifier checks that the contract is in the specified state */
    modifier inState(ProjectState state) {
        require(projectState == state, "Invalid contract state to run function");
        _;
    }

    /** onlyOwner modifier checks that the caller is the owner */
    modifier onlyOwner() {
        require(msg.sender == owner, "You are not the owner");
        _;
    }

    /** enoughFunds modifier checks that the value of the message is greater than or equal to
         the price of the NFT */
    modifier enoughFunds() {
        require(msg.value >= price, "Not enough ether sent");
        _;
    }

    /** constructor function is called when the contract is deployed. It sets the owner to
         the deployer's address and the projectState to emptyVault */
    constructor() {
        owner = payable(msg.sender);
        projectState = ProjectState.emptyVault;
    }

    /** ...functions... */
}
```

Functions for contract: NFTListing

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

contract ERC721Holder {/** ...... */}

contract NFTListing is ERC721Holder {

    /** ...variables, modifiers, constructor... (previous example contract) */

    /** depositAsset function allows the owner to deposit an ERC721 token into the contract.
         It can only be called if the contract is in the emptyVault state and the caller is
         the owner */
    function depositAsset(
        address _NFTAddress,
        uint256 _TokenID
    ) public override onlyOwner inState(ProjectState.emptyVault) {
        super.depositAsset(_NFTAddress, _TokenID);
        projectState = ProjectState.fullVault;
    }

    /** initSale function allows the owner to put the NFT up for sale at a specified price it
        can only be called if the contract is in the fullVault state and the caller is the
        owner */
    function initSale(
        uint256 _price
    ) public onlyOwner inState(ProjectState.fullVault) {
        buyer = address(0);
        price = _price;
        projectState = ProjectState.onSale;
    }

    /** initExclusiveSale function allows the owner to put the NFT up for sale exclusively to
        a specific buyer at a specified price it can only be called if the contract is in the
        fullVault state and the caller is the owner */
    function initExclusiveSale(
        address _buyer,
        uint256 _price
    ) public onlyOwner inState(ProjectState.fullVault)
    {
        buyer = _buyer;
        price = _price;
        projectState = ProjectState.onSale;
    }

    /** cancelSale function allows the owner to cancel the sale of the NFT it can only be
        called if the contract is in the onSale state and the caller is the owner */
    function cancelSale() public onlyOwner inState(ProjectState.onSale) {
        buyer = address(0);
        price = 0;
        /** withdraw the NFT from the contract and set the nftAddress and tokenID variables
            to 0 */
        super.withdrawAsset(owner);
        projectState = ProjectState.emptyVault;
    }

    /** purchase function allows a buyer to purchase the NFT by sending the required amount
        of ether to the contract it can only be called if the contract is in the onSale state
        and the value of the message is greater than or equal to the price of the NFT */
    function purchase() public payable enoughFunds inState(ProjectState.onSale) {
        /** if the sale is exclusive and the caller is not the specified buyer, the function
            should throw an error */
        require((buyer != address(0) && msg.sender != buyer), "Exclusive sale, transaction
            sender isnt buyer");
        /** transfer the ether from the contract to the owner */
        owner.transfer(msg.value);
        /** withdraw the NFT from the contract and transfer it to the caller */
        super.withdrawAsset(msg.sender);
        projectState = ProjectState.emptyVault;
    }
}
```

1.6.4 Understanding our contract

NFTListing is a Solidity contract that is a subclass of the ERC721Holder contract, which means it inherits all the properties and functions of the `ERC721Holder` contract.

In addition to the inherited properties and functions, `NFTListing` has its own set of variables and functions. It has an `enum` 1.2.3 called `ProjectState` with three possible values: `emptyVault`, `fullVault`, and `onSale`. This is an approach we have not used previously in the report, but using an enum is helpful to track the state of the contract, whether it is empty, full, or on sale.

It also has a public address variable called `owner` that stores the owner's address, a public address variable called `buyer` that stores the address of the current buyer (if any), and a public uint256 variable called `price` that stores the listing price of the NFT being sold.

`NFTListing` has three modifiers 1.2.5: `inState`, `onlyOwner`, and `enoughFunds`. The `inState` modifier checks that the contract is in the specified state, the `onlyOwner` modifier checks that the caller is the owner, and the `enoughFunds` modifier checks that the value of the message is greater than or equal to the price of the NFT.

The contract has four functions: `constructor, depositAsset, initSale,` and `purchase`. The constructor function is called when the contract is deployed and sets the `owner` variable to the deployer's address and the `projectState` variable to `emptyVault`.

The `depositAsset` function allows the owner to deposit an ERC721 token into the contract, and it can only be called if the contract is in the `emptyVault` state and the caller is the owner.

The `initSale` function allows the owner to put the NFT up for sale at a specified price, and it can only be called if the contract is in the `fullVault` state and the caller is the owner.

The `purchase` function allows a buyer to purchase the NFT by sending the required amount of ether to the contract, and it can only be called if the contract is in the `onSale` state and the value of the message is greater than or equal to the price of the NFT.

1.6.5 Deploying our application

There are a couple approaches we can take to deploy our contract, for our purposes we will use the Remix Ethereum IDE with a built-in test blockchain.

For completed code, refer to this project's repository: /src/Solidity/. In this directory can be found the full `NFTListing` contract we have written above, along with the following NFT contract which can be used to test the contract:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Counters.sol";


contract BasicNFT is ERC721URIStorage {
    using Counters for Counters.Counter;
    Counters.Counter private _tokenIds;

    constructor() ERC721("BasicNFT", "BNFT") {}

    function mint(string memory tokenURI) public {
        _tokenIds.increment();
        uint256 newItemId = _tokenIds.current();

        _safeMint(msg.sender, newItemId);
        _setTokenURI(newItemId, tokenURI);
    }
}
```

Explore the left menu bar within the Remix IDE. This is a command center for executing contracts and commands on a spoof blockchain.

In the `File Explorer` panel, click the `+` button to create a new file. In the new file, write or paste our Solidity contract code.

Compile and deploy both contracts using the respective IDE left menu bar panels. The IDE is very self explanatory and I urge you to try and test the functionality of our application. Note: ERC721 tokens must be approved for transfer, meaning the contract address of `NFTListing` should be explicitly approved within the individual NFT contract prior to an attempt to interact with the listing contract, see Section 1.5.1.

1.7 Critical Appraisal

Analyzing a programming language such as Solidity has been extremely interesting, and there has been information from [PL] that has proven very insightful to the workings of this language. When I started working on this project, I had little experience to writing Solidity and my exposure to the language was very surface level. My goal for this project is to put in one place all the core aspects and foundational knowledge of Solidity. For reference, I have been unable to find a clear, consise tutorial that starts at the basics, and works all the way through deeper concepts. The online tutorials are typically all for beginners, and I needed a tutorial that someone more advanced than that would also value.

References

[PL] Programming Languages 2022, Chapman University, 2022.

[PR] Project repository Programming Languages 2022, Chapman University, 2022.

[EIP-20] EIP-20 Token Standard, ethereum.org

[Solidity Language Documentation] solidity.readthedocs.io

[Open Zeppelin's solidity library] OpenZepplin contract library

[EIP] Ethereum Improvement Proposals