Introducing Solidity

From this chapter, we will embark on a journey: learning the Solidity language. The previous two chapters introduced blockchains, Ethereum, and their toolsets. Some important concepts related to blockchains, which are essential for having a better understanding and writing efficient code in Solidity, were also discussed. There are multiple languages that target EVM. Some of them are deprecated and others are used with varying degrees of acceptance. Solidity is by far the most popular language for EVM. From this chapter onward, the book will focus on Solidity and its concepts, as well as constructs to help write efficient smart contracts.

In this chapter, we will jump right into understanding Solidity, its structure, data types, and variables. We will cover the following topics in this chapter:

- Solidity and Solidity files
- Structure of a contract
- Data types in Solidity
- Storage and memory data locations
- Literals
- Integers
- Boolean
- The byte data type
- Arrays
- Structure of an array
- Enumeration
- Address
- Mappings

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

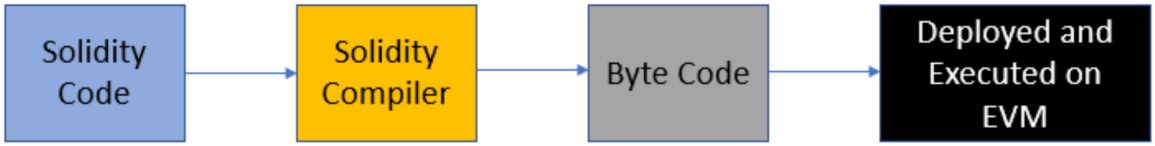
Ethereum Virtual Machine

Solidity is a programming language targeting Ethereum Virtual Machine (EVM). Ethereum blockchain helps extend its functionality by writing and executing code known as smart contracts. We will get into the details of smart contracts in subsequent chapters, but for now, it is enough to know that smart contracts are similar to object-oriented classes written in Java or C++.

EVM executes code that is part of smart contracts. Smart contracts are written in Solidity; however, EVM does not understand the high-level constructs of Solidity. EVM understands lower-level instructions called **bytecode**.

Solidity code needs a compiler to take its code and convert it into bytecode that is understandable by EVM. Solidity comes with a compiler to do this job, known as the Solidity compiler or solc. We downloaded and installed the Solidity compiler in the last chapter using the Node.js npm command.

The entire process is shown in the following diagram, from writing code in Solidity to executing it in EVM:



We have already explored our first Solidity code in the last chapter, when writing our `HelloWorld` contract.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Solidity and Solidity files

Solidity is a programming language that is very close to JavaScript. Similarities between JavaScript and C can be found within Solidity. Solidity is a statically-typed, case-sensitive, and **object-oriented programming** (**OOP**) language. Although it is object-oriented, it supports limited objected orientation features. What this means is that variable data types should be defined and known at compile time. Functions and variables should be written in OOP same way as they are defined. In Solidity, Cat is different from CAT, cat, or any other variation of cat. The statement terminator in Solidity is the semicolon: `;`.

Solidity code is written in Solidity files that have the extension `.sol`. They are human-readable text files that can be opened as text files in any editor including Notepad.

A Solidity file is composed of the following four high-level constructs:

- Pragma
- Comments
- Import
- Contracts/library/interface

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Pragma

Pragma is generally the first line of code within any Solidity file. `pragma` is a directive that specifies the compiler version to be used for current Solidity file.

Solidity is a new language and is subject to continuous improvement on an on-going basis. Whenever a new feature or improvement is introduced, it comes out with a new version. The current version at the time of writing was 0.4.19.

With the help of the `pragma` directive, you can choose the compiler version and target your code accordingly, as shown in the following code example:

```
pragma Solidity ^0.4.19;
```

Although it is not mandatory, it is a good practice to declare the `pragma` directive as the first statement in a Solidity file.

The syntax for the `pragma` directive is as follows:

```
pragma Solidity <<version number>> ;
```

Also notice the case-sensitivity of the directive. Both `pragma` and `Solidity` are in small letters, with a valid version number and statement terminated with a semicolon.

The version number comprises of two numbers—a **major build** and a **minor build** number.

The major build number in the preceding example is `4` and the minor build number is `19`. Generally, there are fewer or no breaking changes within minor versions but there could be significant changes between major versions. You should choose a version that best suits your requirements.

The `^` character, also known as **caret**, is optional in version numbers but plays a significant role in deciding the version number based on the following rules:

- The `^` character refers to the latest version within a major version. So, ^0.4.0 refers to the latest version within build number 4, which currently would be 0.4.19.
- The `^` character will not target any other major build apart from the one that is provided.
- The Solidity file will compile only with a compiler with 4 as the major build. It will not compile with any other major build.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

As a good practice, it is better to compile Solidity code with an exact compiler version rather than using `^`. There are changes in newer version that could deprecate your code while using `^` in `pragma`. For example, the `throw` statement got deprecated and newer constructs such as `assert`, `require`, and `revert` were recommended for use in newer versions. You do not want to get surprised on a day when your code starts behaving differently.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Comments

Any programming language provides the facility to comment code and so does Solidity. There are the following three types of comment in Solidity:

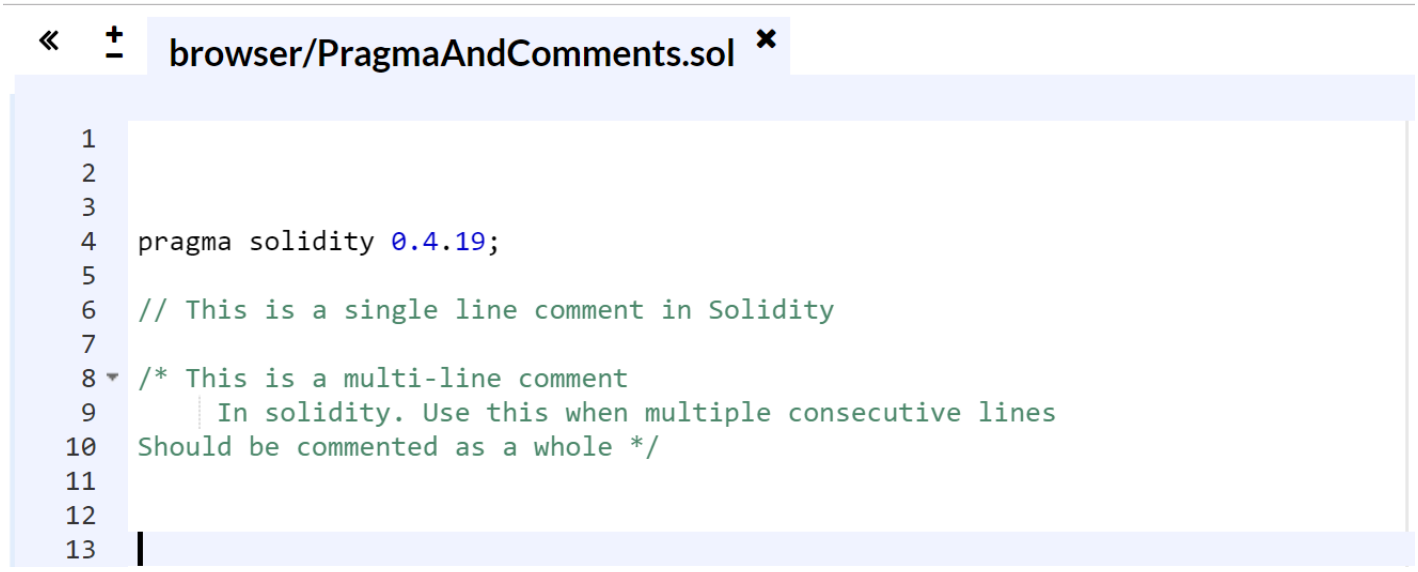
- Single-line comments
- Multiline comments
- **Ethereum Natural Specification** (**Natspec**)

Single-line comments are denoted by a double forward slash `//`, while multiline comments are denoted using `/*` and `*/`. Natspec has two formats: `///` for single-line and a combination of `/**` for beginning and `*/` for end of multiline comments. Natspec is used for documentation purposes and it has its own specification. The entire specification is available at https://github.com/ethereum/wiki/wiki/Ethereum-Natural-Specification-Format.

Let's take a look at Solidity comments in the following code:

```
// This is a single-line comment in Solidity
/* This is a multiline comment
In Solidity. Use this when multiple consecutive lines
Should be commented as a whole */
```

In Remix, the `pragma` directive and comments are as shown in the following screenshot:



Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The import statement

The `import` keyword helps import other Solidity files and we can access its code within the current Solidity file and code. This helps us write modular Solidity code.

The syntax for using `import` is as follows:

```
import <<filename>> ;
```

File names can be fully explicit or implicit paths. The forward slash `/` is used for separating directories from other directories and files while `.` is used to refer to the current directory and `..` is used to refer to the parent directory. This is very similar to the Linux bash way of referring to a file. A typical `import` statement is shown here. Also, note the semicolon towards the end of the statement in the following code:

```
import 'CommonLibrary.sol';
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Contracts

Apart from `pragma`, `import`, and comments, we can define contracts, libraries, and interfaces at the global or top level. We will explore contracts, libraries, and interfaces in depth in subsequent chapters. This chapter assumes that you understand that multiple contracts, libraries, and interfaces can be declared within the same Solidity file. The `library`, `contract`, and `interface` keywords shown in the following screenshot are case-sensitive in nature:

```
//contracts.sol

pragma solidity 0.4.19;

// This is a single line comment in Solidity

/* This is a multi-line comment
    In solidity. Use this when multiple consecutive lines
Should be commented as a whole */

contract firstContract {

}

contract secondContract {

}

library stringLibrary {

}

library mathLibrary {

}

interface IBank{

}

interface IAccount {

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Structure of a contract

The primary purpose of Solidity is to write smart contracts for Ethereum. Smart contracts are the basic unit of deployment and execution for EVMs. Although multiple chapters later in this book are dedicated to writing and developing smart contracts, the basic structure of smart contracts is discussed in this chapter.

Technically, smart contracts are composed of two constructs—variables and functions. There are multiple facets to both variables and functions and that is again something that will be discussed throughout this book. This section is about describing the general structure of a smart contract using the Solidity language.

A contract consists of the following multiple constructs:

- State variables
- Structure definitions
- Modifier definitions
- Event declarations
- Enumeration definitions
- Function definitions

A typical contract consists of all the preceding constructs. In the following screenshot, it is to be noted that each of these constructs in turn consists of multiple other constructs, which will be discussed in subsequent chapter when these topics are discussed in detail:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables
    int public stateIntVariable; // vriable of integer type
    string stateStringVariable; //variable of string type
    address personIdentifier; // variable of address type
    myStruct human; // variable of structure type
    bool constant hasIncome = true; //variable of constant nature

    //structure definition
    struct myStruct {
        string name; //variable fo type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    //modifier declaration
    modifier onlyBy(){
        if (msg.sender == personIdentifier) {
            _;
        }
    }

    // event declaration
    event ageRead(address, int );

    //enumeration declaration
    enum gender {male, female}

    //function definition
    function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

       human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

       gender _gender = gender.male; //using enum

       ageRead(personIdentifier, stateIntVariable);
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

State variables

Variables in programming refer to storage location that can contain values. These values can be changed during runtime. The variable can be used at multiple places within code and they will all refer to the value stored within it. Solidity provides two types of variable—state and memory variables. In this section, we will introduce state variables.

One of the most important aspects of Solidity contracts is state variables. It is these state variables that are permanently stored in a blockchain/Ethereum ledger by miners. Variables declared in a contract that are not within any function are called **state variables**. State variables store the current values of the contract. The allocated memory for a state variable is statically assigned and it cannot change (the size of memory allocated) during the lifetime of the contract. Each state variable has a type that must be defined statically. The Solidity compiler must ascertain the memory allocation details for each state variables and so the state variable data type must be declared.

State variables also have additional qualifiers associated with them. They can be any one of the following:

- `internal`: By default, the state variable has the `internal` qualifier if nothing is specified. It means that this variable can only be used within current contract functions and any contract that inherits from them. These variables cannot be accessed from outside for modification; however, they can be viewed. An example of internal state variable is as follows:

```
int internal StateVariable ;
```

- `private`: This qualifier is like `internal` with additional constraints. Private state variables can only be used in contracts declaring them. They cannot be used even within derived contracts. An example of a private state variable is as follows:

```
int private privateStateVariable ;
```

- `public`: This qualifier makes state variables access directly. The Solidity compiler generates a `getter` function for each public state variable. An example of a public state variable is as follows:

```
int public stateIntVariable ;
```

- `constant`: This qualifier makes state variables immutable. The value must be assigned to the variable at declaration time itself. In fact, the compiler will replace references of this variable everywhere in code with the assigned value. An example of a constant state variable is as follows:

```
bool constant hasIncome = true;
```

As mentioned previously, each state variable has an associated data type. A data type helps us determine the memory requirements for the variable and ascertain the values that can be stored in them. For example, a state variable of type `uint8` also known as **unsigned integer** is allocated a predetermined memory size and it

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

can contain values ranging from `0` to `255`. Any other value is regarded as foreign and is not acceptable by compiler and runtime for storing it in this variable.

Solidity provides the following multiple out-of-box data types:

- `bool`
- `uint/int`
- `bytes`
- `address`
- `mapping`
- `enum`
- `struct`
- `bytes/String`

Using `enum` and `struct`, it is possible to declare custom user-defined data types as well. Later in this chapter, a complete section has been dedicated to data types and variables.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Structure

Structures or structs helps implement custom user-defined data types. A structure is a composite data type, consisting of multiple variables of different data types. They are very similar to contracts; however, they do not contain any code within them. They consist of only variables.

There are times when you would like to store related data together. Suppose you want to store information about an employee, say the employee name, age, marriage status, and bank account numbers. To represent this, these individual variables related to single employee, a structure in Solidity can be declared using the `struct` keyword. The variables within a structure are defined within opening and closing `{}` brackets as shown in the following screenshot:

```
//structure definition
struct myStruct {
    string name; //variable fo type string
    uint myAge; // variable of unsigned integer type
    bool isMarried; // variable of boolean type
    uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
}
```

To create an instance of a structure, the following syntax is used. There is no need to explicitly use the `new` keyword. The `new` keyword can only be used to create an instance of contracts or arrays as shown in the following screenshot:

```
human = myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct
```

Multiple instance of `struct` can be created in functions. Structs can contain array and the `mapping` variables, while mappings and arrays can store values of type `struct`.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Modifiers

In Solidity, a modifier is always associated with a function. A modifier in programming languages refers to a construct that changes the behavior of the executing code. Since a modifier is associated with a function in Solidity, it has the power to change the behavior of functions that it is associated with. For easy understanding of modifiers, think of them as a function that will be executed before execution of the target function. Suppose you want to invoke the `getAge` function but, before executing it, you would like to execute another function that could check the current state of the contract, values in incoming parameters, the current value in state variables, and so on and accordingly decide whether the target function `getAge` should be executed. This helps in writing cleaner functions without cluttering them with validation and verification rules. Moreover, the modifier can be associated with multiple functions. This ensures cleaner, more readable, and more maintainable code.

A modifier is defined using the `modifier` keyword followed by the `modifier` identifier, any parameters it should accept, and then code within the `{}` brackets. An _ underscore in a modifier means: execute the target function. You can think of this as the underscore being replaced by the target function inline. `payable` is an out-of-the-box modifier provided by Solidity which when applied to any function allows that function to accept Ether.

A `modifier` keyword is declared at the contract level, as shown in the following screenshot:

```
//modifier declaration
modifier onlyBy(){
    if (msg.sender == personIdentifier) {
        _;
    }
}
```

As we can see, in the preceding screenshot of the code snippet, a modifier named `onlyBy()` is declared at the contract level. It checks the value of the incoming address using `msg.sender` with an address stored in the state variable. Some things such as `msg.sender` might not be understandable to readers; we will cover these in depth in the next chapter.

The modifier is associated with a `getAge` function as shown in the following screenshot:

```
//function definition
function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

   human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

   gender _gender = gender.male; //using enum
}
```

The `getAge` function can only be executed by an account that has the same address as that stored in the

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

contract's `_personIdentifier` state variable. The function will not be executed if any other account tries to invoke it.

It is to be noted that anybody can invoke the `getAge` function, but execution will only happen for single a account.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Events

Solidity supports events. Events in Solidity are just like events in other programming languages. Events are fired from contracts such that anybody interested in them can trap/catch them and execute code in response. Events in Solidity are used primarily for informing the calling application about the current state of the contract by means of the logging facility of EVM. They are used to notify applications about changes in contracts and applications can use them to execute their dependent logic. Instead of applications they keep polling the contract for certain state changes; the contract can inform them by means of events.

Events are declared within the contract at the global level and invoked within its functions. An event is declared using the `event` keyword, followed by an identifier and parameter list and terminated with a semicolon. The values in parameters can be used to log information or execute conditional logic. Event information and its values are stored as part of transactions within blocks. In the last chapter, while discussing the properties of a transaction, a property named `LogsBloom` was introduced. Events raised as part of a transaction are stored within this property.

There is no need to explicitly provide parameter variables—only data types are sufficient as shown in the following screenshot:

```
// event declaration
event ageRead(address, int );
```

An event can be invoked from any function by its name and by passing the required parameters, as shown in the following screenshot:

```
/function definition
unction getAge (address _personIdentifier) onlyBy() payable external returns (uint)

 human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

 gender _gender = gender.male; //using enum

 ageRead(personIdentifier, stateIntVariable);
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Enumeration

The `enum` keyword is used to declare enumerations. Enumerations help in declaring a custom user-defined data type in Solidity. `enum` consists of an enumeration list, a predetermined set of named constants.

Constant values within an `enum` can be explicitly converted into integers in Solidity. Each constant value gets an integer value, with the first one having a value of 0 and the value of each successive item is increased by 1.

An `enum` declaration uses the `enum` keyword followed by enumeration identifier and a list of enumeration values within the `{}` brackets. It is to be noted that an `enum` declaration does not have a semicolon as its terminator and that there should be at least one member declared in the list.

An example of `enum` is as follows:

```
enum gender {male, female}
```

A variable of enumeration can be declared and assigned a value as shown in the following code:

```
gender _gender = gender.male ;
```

It is not mandatory to define `enum` in a Solidity contract. `enum` should be defined if there is a constant list of items that do not change like the example shown previously. These become good example for an `enum` declaration. They help make your code more readable and maintainable.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Functions

Functions are the heart of Ethereum and Solidity. Ethereum maintains the current state of state variables and executes transaction to change values in state variables. When a function in a contract is called or invoked, it results in the creation of a transaction. Functions are the mechanism to read and write values from/to state variables. Functions are a unit of code that can be executed on-demand by calling it. Functions can accept parameters, execute its logic, and optionally return values to the caller. They can be named as well as anonymous. Solidity provides named functions with the possibility of only one unnamed function in a contract called the fallback function. We will know more about fallback functions later in the book.

The syntax for declaring functions in Solidity is as follows:

```
//function definition
function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {


}
```

A function is declared using the `function` keyword followed by its identifier—`getAge`, in this case. It can accept multiple comma-separated parameters. The parameter identifiers are optional, but data types should be provided in the parameter list. Functions can have modifiers attached, such as `onlyBy()` in this case.

There are a couple of additional qualifiers that affect the behavior and execution of a function. Functions have visibility qualifiers and qualifiers, related to what actions can be executed within the function. Both visibility and function ability-related keywords are discussed next. Functions can also return data and this information is declared using the `return` keyword, followed by list of return parameters. Solidity can return multiple parameters.

Functions has visibility qualifier associated with them similar to state variables. The visibility of a function can be any one of the following:

- `public`: This visibility makes function access directly from outside. They become part of the contracts interface and can be called both internally and externally.
- `internal`: By default, the state variable has `internal` qualifier if nothing is specified. It means that this function can only be used within the current contract and any contract that inherits from it. These functions cannot be accessed from outside. They are not part of the contracts interface.
- `private`: Private functions can only be used in contracts declaring them. They cannot be used even within derived contracts. They are not part of the contracts interface.
- `external`: This visibility makes function access directly from externally but not internally. These functions become part of the contracts interface.

Functions can also have the following additional qualifiers that change their behavior in terms of having the ability to change contract state variables:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

- `constant`: These functions do not have the ability to modify the state of blockchain. They can read the state variables and return back to the caller, but they cannot modify any variable, invoke an event, create another contract, call other functions that can change state, and so on. Think of `constant` functions as functions that can read and return current state variable values.
- `view`: These functions are aliases of constant functions.
- `pure`: Pure functions further constraints the ability of functions. Pure functions can neither read and write—in short, they cannot access state variables. Functions that are declared with this qualifier should ensure that they will not access the current state and transaction variables.
- `payable`: Functions declared with the `payable` keyword has ability to accept Ether from the caller. The call will fail in case Ether is not provided by sender. A function can only accept Ether if it is marked as `payable`.

We will discuss the preceding qualifiers in detail in subsequent chapters.

Functions can be invoked by their names.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Data types in Solidity

Solidity data types can broadly be classified in the following two types:

- Value types
- Reference types

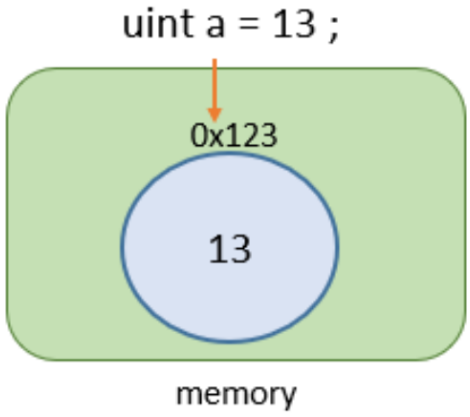
These two types in Solidity differ based on the way they are assigned to a variable and stored in EVM. Assigning a variable to another variable can be done by creating a new copy or just by coping the reference. Value types maintains independent copies of variables and changing the value in one variable does not effect value in another variable. However, changing values in reference type variables ensures that anybody referring to that variables gets updates value.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Value types

A type is referred as value type if it holds the data (value) directly within the memory owned by it. These types have values stored with them, instead of elsewhere. The same is illustrated in following diagram. In this example, a variable of data type **unsigned integer** (**uint**) is declared with **13** as its data(value). The variable **a** has memory space allocated by EVM which is referred as **0x123** and this location has the value **13** stored. Accessing this variable will provide us with the value **13** directly:



Value types are types that do not take more than 32 bytes of memory in size. Solidity provides the following value types:

- `bool`: The boolean value that can hold true or false as its value
- `uint`: These are unsigned integers that can hold 0 and positive values only
- `int`: These are signed integers that can hold both negative and positive values
- `address`: This represents an address of an account on Ethereum environment
- `byte`: This represents fixed sized byte array (`byte1` to `bytes32`)
- `enum`: Enumerations that can hold predefined constant values

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Passing by value

When a value type variable is assigned to another variable or when a value type variable is sent as an argument to a function, EVM creates a new variable instance and copies the value of original value type into target variable. This is known as passing by value. Changing values in original or target variables will not affect the value in another variable. Both the variables will maintain their independent, isolated values and they can change without the other knowing about it.

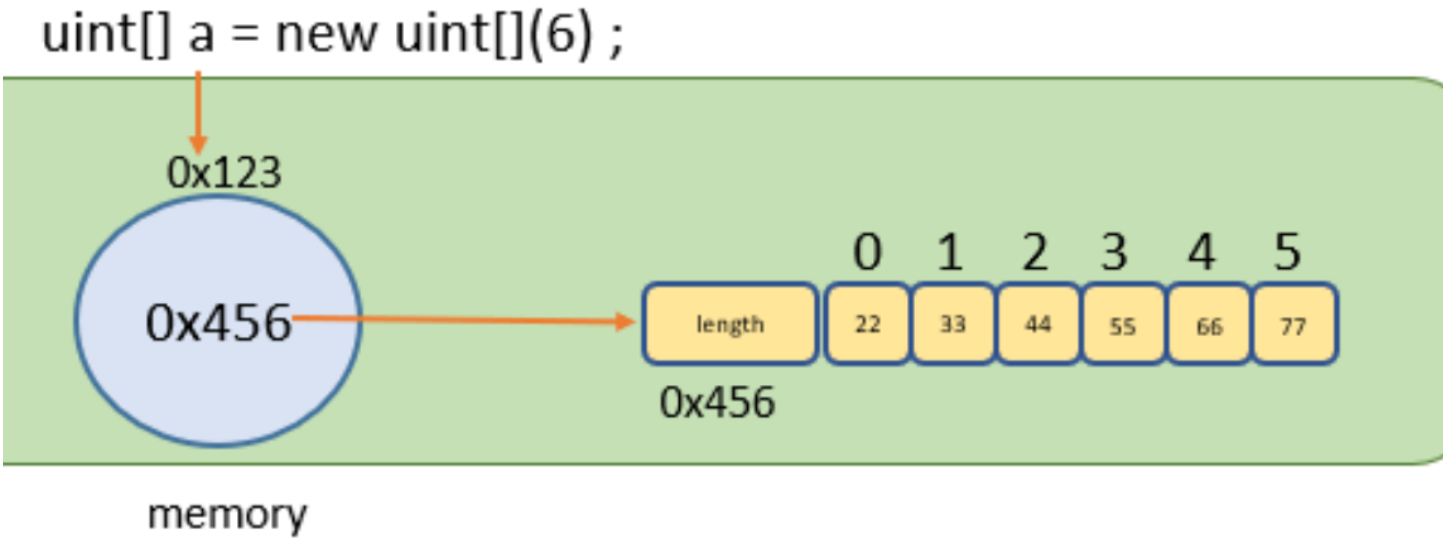
Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Reference types

Reference types, unlike value types, do not store their values directly within the variabless themselves. Instead of the value, they store the address of the location where the value is stored. The variable holds the pointer to another memory location that holds the actual data. Reference types are types that can take more than 32 bytes of memory in size. Reference types are shown next, by means of an illustration.

In the following example, an array variable of data type **uint** is declared with size **6**. Arrays in Solidity are based at zero and so this array can hold seven elements. The variable **a** has memory space allocated by EVM which is referred as **0x123** and this location has a pointer value **0x456** stored in it. This pointer refers to the actual memory location where the array data is stored. When accessing the variable, EVM dereferences the value of the pointer and shows the value from the array index as shown in the following diagram:



Solidity provides the following reference types:

- **Arrays**: These are fixed as well as dynamic arrays. Details are given later in this chapter.
- **Structs**: These are custom, user-defined structures.
- **String**: This is sequence of characters. In Solidity, strings are eventually stored as bytes. Details are give later in this chapter.
- **Mappings**: This is similar to a hash table or dictionary in other languages storing key-value pairs.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Passing by reference

When a reference type variable is assigned to another variable or when a reference type variable is sent as an argument to a function, EVM creates a new variable instance and copies the pointer from the original variable into the target variable. This is known as passing by reference. Both the variables are pointing to the same address location. Changing values in the original or target variables will change the value in other variables also. Both the variables will share the same values and change committed by one is reflected in the other variable.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Storage and memory data locations

Each variable declared and used within a contract has a data location. EVM provides the following four data structures for storing variables:

- Storage: This is global memory available to all functions within the contract. This storage is a permanent storage that Ethereum stores on every node within its environment.
- Memory: This is local memory available to every function within a contract. This is short lived and fleeting memory that gets torn down when the function completes its execution.
- Calldata: This is where all incoming function execution data, including function arguments, is stored. This is a non-modifiable memory location.
- Stack: EVM maintains a stack for loading variables and intermediate values for working with Ethereum instruction set. This is working set memory for EVM. A stack is 1,024 levels deep in EVM and if it store anything more than this it raises an exception.

The data location of a variable is dependent on the following two factors:

- Location of variable declaration
- Data type of the variable

Based on the preceding two factors, there are rules that govern and decide the data location of a variable. The rules are mentioned here. Data locations also effect the way assignment operator works. Both assignment and data locations are explained by means of rules that govern them.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 1

Variables declared as state variables are always stored in the storage data location.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 2

Variables declared as function parameters are always stored in the memory data location.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 3

Variables declared within functions, by default, are stored in memory data location. However, there are following few caveats:

- The location for value type variables is memory within a function while the default for a reference type variable is storage.

Please note that storage is the default for reference type variable declared within a function. However, it can be overridden.

- By overriding the default location, reference types variables can be located at the memory data location. The reference types referred are arrays, structs, and strings.
- Reference types declared within a function without being overridden should always point to a state variable.
- Value type variables declared in a function cannot be overridden and cannot be stored at the storage location.
- Mappings are always declared at storage location. This means that they cannot be declared within a function. They cannot be declared as memory types. However, mappings in a function can refer to mappings declared as state variables.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 4

Arguments supplied by callers to function parameters are always stored in a calldata data location.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 5

Assignments to state variable from another state variable always creates a new copy. Two value type state variables `stateVar1` and `stateVar2` are declared. Within the `getUInt` function, `stateVar2` is assigned to `stateVar1`. At this stage, the values in both the variables are `40`. The next line of code changes the value of `stateVar2` to `50` and returns `stateVar1`. The returned value is `40` illustrating that each variable maintains its own independent value a shown in the following screenshot:

```
pragma solidity 0.4.19;

contract DemoStoragetoStorageValueTypeAssignment {

    uint stateVar1 = 20;

    uint stateVar2 = 40;

    function getUInt() returns (uint)
    {
        stateVar1 = stateVar2;

        stateVar2 = 50;

        return stateVar1; // returns 40

    }

}
```

Two array type state variables, `stateArray1` and `stateArray2`, are declared. Within the `getUInt` function, `stateArray2` is assigned to `stateArray1`. At this stage, the values in both the variables are the same. The next line of code changes one of the values in `stateArray2` to `5` and returns the element at same location from the `stateArray1` array. The returned value is `4`, illustrating that each variable maintains its own independent value as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract DemoStoragetoStorageReferenceTypeAssignment {

    uint[2] stateArray1 = [uint(1), 2];

    uint[2] stateArray2 = [uint(3), 4];

    function getUInt() returns (uint)
    {
      stateArray1 = stateArray2;

      stateArray2[1] = 5;

      return stateArray1[1]; // returns 4

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 6

Assignments to storage variables from another memory variable always create a new copy.

A fixed array of uint `stateArray` is declared as a state variable. Within the `getUInt` function a local memory located fixed array of uint `localArray` is defined and initialized. The next line of code assigns `localArray` to `stateArray`. At this stage, the values in both the variables are the same. The next line of code changes one of the values in `localArray` to `10` and returns the element at same location from the `stateArray1` array. The returned value is `2`, illustrating that each variable maintains its own independent value as shown in the following screenshot:

```
pragma solidity 0.4.19;

contract DemoMemorytoStorageReferenceTypeAssignment {

    uint[2] stateArray ;
    function getUInt() returns (uint)
    {
      uint[2] memory localArray = [uint(1), 2];

      stateArray = localArray;

      localArray[1] = 10;

      return stateArray[1]; // returns 2

    }

}
```

A value type `stateVar` state variables is declared and initialized with value `20`. Within the `getUInt` function, a `localVar` local variable is declared with value `40`. In next line of code, the `localVar` local variable is assigned to `stateVar`. At this stage, the values in both the variables are `40`. The next line of code changes the value of `localVar` to `50` and returns `stateVar`. The returned value is `40`, illustrating that each variable maintains its own independent value as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract DemoMemorytoStorageValueTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
      uint localVar = 40;

      stateVar = localVar;

      localVar = 50;

      return stateVar; // returns 40

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 7

Assignments to memory variable from another state variable always creates a new copy. A value type state variable, `stateVar` is declared and initialized with value `20`. Within the `getUInt` function a local variable of type `uint` is declared and initiated with value `40`. The `stateVar` variable is assigned to the `localVar` variable. At this stage, the values in both the variables are `20`. The next line of code changes the value of `stateVar` to `50` and returns `localVar`. The returned value is `20`, illustrating that each variable maintains its own independent value as shown in the following screenshot:

```
pragma solidity 0.4.19;

contract DemoStoragetoMemoryValueTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
      uint localVar = 40;

      localVar = stateVar;

      stateVar = 50;

      return localVar; // returns 20

    }

}
```

A fixed array of uint `stateArray` is declared as state variable. Within the `getUInt` function, a local memory located, fixed array of uint `localArray` is defined and initialized with the `stateArray` variable. At this stage, the values in both the variables are the same. The next line of code changes one of the values in `stateArray` to `5` and returns the element at the same location from the `localArray1` array. The returned value is `2`, illustrating that each variable maintains its own independent value as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract DemoStoragetoMemoryReferenceTypeAssignment {

    uint[2] stateArray = [uint(1), 2];
    function getUInt() returns (uint)
    {
      uint[2] memory localArray = stateArray;

      stateArray[1] = 5;

      return localArray[1]; // returns 2

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Rule 8

Assignments to a memory variable from another memory variable do not create a copy for reference types; however, they do create a new copy for value types. The code listing shown in the following screenshot illustrates that value type variables in memory are copied by value. The value of `localVar1` is not affected by change in value of the `localVar2` variable:

```
pragma solidity 0.4.19;

contract DemoMemorytoMemoryValueTypeAssignment {


    function getUInt() returns (uint)
    {
      uint localVar1 = 40;

      uint localVar2 = 80;

      localVar1 = localVar2;

      localVar2 = 100;

      return localVar1; // returns 80

    }

}
```

The code listing shown in the following screenshot illustrates that reference type variables in memory are copied by reference. The value of `otherVar` is affected by change in the `someVar` variable:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract DemoMemorytoMemoryReferenceTypeAssignment {

    uint stateVar = 20;
    function getUInt() returns (uint)
    {
        uint[] memory someVar = new uint[](1);

        someVar[0] = 23;

        uint[] memory  otherVar = someVar;

        someVar[0] = 45;

        return (otherVar[0]); //returns 45

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Literals

Solidity provides usage of literal for assignments to variables. Literals do not have names; they are the values themselves. Variables can change their values during a program execution, but a literal remains the same value throughout. Take a look at the following examples of various literals:

- Examples of integer literal are 1, 10, 1,000, -1, and -100.
- Examples of string literals are "Ritesh" and 'Modi'. String literals can be in single or double quotes.
- Examples of address literals are 0xca35b7d915458ef540ade6068dfe2f44e8fa733c and 0x1111111111111111111111111111111111111111.
- Hexadecimal literals are prefixed with the `hex` keyword. An example of hexadecimal literals is hex"1A2B3F".
- Solidity supports decimal literals with use of dot. Examples include 4.5 and 0.2.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Integers

Integers help in storing numbers in contracts. Solidity provides the following two types of integer:

- **Signed integers**: Signed integers can hold both negative and positive values.
- **Unsigned integers**: Unsigned integers can hold only positive values along with zero. They can also hold negative values apart from positive and zero values.

There are multiple flavors of integers in Solidity for each of these types. Solidity provides uint8 type to represent 8 bit unsigned integer and thereon in multiples of 8 till it reaches 256. In short, there could be 32 different declarations of uint with different multiples of of 8, such as uint8, uint16, unit24, as far as uint256 bit. Similarly, there are equivalent data types for integers such as int8, int16 till int256.

Depending on requirements, an appropriately sized integer should be chosen. For example, while storing values between 0 and 255 uint8 is appropriate, and while storing values between -128 to 127 int8 is more suitable. For higher values, larger integers can be used.

The default value for both signed and unsigned integers is zero, to which they are initialized automatically at the time of declaration. Integers are value types; however, when used as an array they are referred as reference types.

Mathematical operations such as addition, subtraction, multiplication, division, exponential, negation, post-increment, and pre-increment can be performed on integers. The following screenshot shows some of these examples:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract AllAboutInts {

    uint  stateUInt = 20 ; //state variable
    uint  stateInt = 20 ; //state variable

    function getUInt(uint incomingValue)
    {
        uint memoryuint = 256 ;
        uint256 memoryuint256 = 256 ;
        uint8 memoryuint8 = 8 ; //can store value from 0 upto 255

        //addition of two uint8
        uint256 result = memoryuint8 + memoryuint ;

        // assignAfterIncrement = 9 and memoryuint8 = 9
       uint8 assignAfterIncrement = ++memoryuint8 ;

        // assignAfterIncrement = 9 and memoryuint8 = 10
      uint8 assignBeforeIncrement = memoryuint8++;


    }

    function getInt(int incomingValue)
    {
        int memoryInt = 256 ;
        int256 memoryInt256 = 256 ;
        int8 memoryInt8 = 8 ; //can store value from -128 to 127
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Boolean

Solidity, like any programming language, provides a boolean data type. The `bool` data type can be used to represent scenarios that have binary results, such as `true` or `false`, `1` or `0`, and so on. The valid values for this data type are `true` and `false`. It is to be noted that bools in Solidity cannot be converted to integers, as they can in other programming languages. It's a value type and any assignment to other boolean variables creates a new copy. The default value for `bool` in Solidity is `false`.

A `bool` data type is declared and assigned a value as shown in the following code:

```
bool isPaid = true;
```

It can be modified within contracts and can be used in both incoming and outgoing parameters and the return value, as shown in the following screenshot:

```
pragma solidity 0.4.19;

contract boolContract {

    bool isPaid = true;

    function manageBool() returns (bool)
    {
      isPaid = false;

      return isPaid; //returns false
    }

   function convertToUint() returns (uint8)
    {
      isPaid = false;

      return uint8(isPaid); //error
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The byte data type

Byte refers to 8 bit signed integers. Everything in memory is stored in bits consisting of binary values—0 and 1. Solidity also provides the byte data type to store information in binary format. Generally, programming languages have a single data type for representing bytes. However, Solidity has multiple flavors of the byte type. It provides data types in the range from `bytes1` to `bytes32` inclusive, to represent varying byte lengths, as required. These are called **fixed sized byte arrays** and are implemented as value types. The `bytes1` data type represents 1 byte and `bytes2` represents 2 bytes. The default value for byte is `0x00` and it gets initialized with this value. Solidity also has a `byte` type that is an alias to `bytes1`.

A byte can be assigned byte values in hexadecimal format, as follows:

```
bytes1 aa = 0x65;
```

A byte can be assigned integer values in decimal format, as follows:

```
bytes1 bb = 10;
```

A byte can be assigned negative integer values in decimal format, as follows:

```
bytes1 ee = -100;
```

A byte can be assigned character values as follows:

```
bytes1 dd = 'a';
```

In the following code snippet, a value of `256` cannot fit in a single byte and needs a bigger byte array:

```
bytes2 cc = 256;
```

The code listing in the following screenshot shows how to store binary, positive, and negative integers, and character literals in fixed sized byte arrays.

We can also perform bitwise operations such as `and`, `or`, `xor`, `not`, and left and right shift operations on the `byte` data type:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract bytesContract {

    bytes1 aa = 0x65;
    bytes1 bb = 10;
    bytes2 cc = 256;
    bytes1 dd = 'a';
    bytes1 ee = -100;

    function getintaa() returns (uint)
    {
      return uint(aa); //returns 101
    }

    function getbyteaa() returns (bytes1)
    {
      return aa; //returns 0x65
    }

    function getbytebb() returns (bytes1)
    {
      return bb; //returns 0x0a
    }

    function getintbb() returns (uint)
    {
      return uint(bb); //returns 10
    }

    function getbytecc() returns (bytes2)
    {
      return cc; //returns 0x0100
    }

    function getintcc() returns (uint)
    {
      return uint(cc); //returns 256
    }

    function getbytedd() returns (bytes2)
    {
      return dd; //returns 0x6100 or 0x61 for bytes1
    }

    function getintdd() returns (uint)
    {
      return uint(dd); //returns 97
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Arrays

Arrays are discussed as data types but, more specifically they are data structures that are dependent on other data types. Arrays refer to groups of values of the same type. Arrays help in storing these values together and ease the process of iterating, sorting, and searching for individuals or subsets of elements within this group. Solidity provides rich array constructs that cater to different needs.

An example of an array in Solidity is as follows:

```
uint[5] intArray ;
```

Arrays in Solidity can be fixed or dynamic.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Fixed arrays

Fixed arrays refer to arrays that have a pre-determined size mentioned at declaration. Examples of fixed arrays are as follows:

```
int[5] age ; // array of int with 5 fixed storage space allocated
byte[4] flags ; // array of byte with 4 fixed storage space allocated
```

Fixed arrays cannot be initialized using the `new` keyword. They can only be initialized inline, as shown in the following code:

```
int[5] age = [int(10), 20,30,40,50] ;
```

They can also be initialized inline within a function later, as follows:

```
int[5] age ;
age = [int(10),2,3,4,5];
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Dynamic arrays

Dynamic arrays refer to arrays that do not have a pre-determined size at the time of declaration; however, their size is determined at runtime. Take a look at the following code:

```
int[] age ; // array of int with no fixed storage space allocated. Storage is allocated du

byte[] flags ; // array of byte with no fixed storage space allocated
```

Dynamic arrays can be initialized inline or using the `new` operator. The initialization can happen at the time of declaration as follows:

```
int[] age = [int(10), 20,30,40,50] ;
int[] age = new int[](5) ;
```

The initialization can also happen within a function later in the following two different steps:

```
int[] age ;
age = new int[](5) ;
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Special arrays

Solidity provides the following two special arrays:

- The bytes array
- The String array

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The bytes array

The `bytes` array is a dynamic array that can hold any number of bytes. It is not the same as `byte []`. The `byte []` array takes 32 bytes for each element whereas `bytes` tightly holds all the bytes together.

Bytes can be declared as a state variable with initial length size as shown in the following code:

```
bytes localBytes = new bytes(0) ;
```

This can be also divided into the following two code lines similar to previously discussed arrays:

```
bytes localBytes ;
localBytes= new bytes (10) ;
```

Bytes can be assigned values directly, as follows:

```
localBytes = "Ritesh Modi";
```

Also, values can be pushed into it, as shown in the following code, if it is located at the storage location:

```
localBytes.push(byte(10));
```

Bytes also provide a read/write `length` property, as follows:

```
return localBytes.length; //reading the length property
```

Take a look at the following code as well:

```
localBytes.length = 4; //setting bytes length to 4 bytes
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The String array

Strings are dynamic data types that are based on bytes arrays discussed in the previous section. They are very similar to bytes with additional constraints. Strings cannot be indexed or pushed and do not have the `length` property. To perform any of these actions on string variables, they should first be converted into bytes and then converted back to strings after the operation.

Strings can be composed of characters within single or double quotes.

Strings can be declared and assigned values directly, as follows:

```
String name = 'Ritesh Modi" ;
```

They can be also converted to bytes, as follows:

```
Bytes byteName = bytes(name) ;
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Array properties

There are basic properties supported by arrays. In Solidity, due to the multiple types of array, not every type supports all of these properties.

These properties are as follows:

- `index`: This property used for reading individual array elements is supported by all types of arrays, except for the string type. The `index` property for writing to individual array element is supported for dynamic arrays, fixed arrays, and the bytes type only. Writing is not supported for string and fixed sized byte arrays.
- `push`: This property is supported by dynamic arrays only.
- `length`: This property is supported by all arrays from read perspective, except for the `string` type. Only dynamic arrays and bytes support modifying the `length` property.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Structure of an array

We have already briefly touched on the topic of structures. Structures help in defining custom user-defined data structures. Structures help in group multiple variables of different data types into a single type. A structure does not contain any programming logic or code for execution; it just contains a variable declaration. Structures are reference types and are treated as complex type in Solidity.

Structures can be defined as state variables, as shown in the next code illustration. A struct composed of `string`, `uint`, `bool`, and `uint` arrays is defined. There are two state variables. They are on the storage location. While the first `stateStructure1` state variable is initialized at the time of declaration, the other `stateStructure1` state variable is left to be initialized later within a function.

A local structure at the memory location is declared and initialized within the `getAge` function.

Another structure is declared that acts as a pointer to the `stateStructure` state variable.

A third local structure is declared that refers to the previously created `localStructure` local structure.

A change in one of the properties of `localStructure` is performed while the previously declared state structure is initialized and finally the age from `pointerLocalStructure` is returned. It returns the new value that was assigned to `localStructure`, as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables

    //structure definition
    struct myStruct {
        string name; //variable fo type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    // state structure
    myStruct  stateStructure = myStruct("Ritesh", 10, true, new uint[](2));

    myStruct  stateStructure1;


    //function definition
    function getAge ()  returns (uint) {

       // local structure
       myStruct memory localStructure = myStruct("Modi", 20 ,false, new uint[](2));

       //local pointer to State structure
       myStruct pointerStructure = stateStructure;

       // pointerlocalStructure is reference to localStructure
       myStruct memory pointerlocalStructure = localStructure;

       //changing value in localStructure
       localStructure.myAge = 30;

       //assigning values to state variable
       stateStructure1 =   myStruct("Ritesh", 10, true, new uint[](2));

       //returning pointerlocalStructure.Age -- returns 30
       return pointerlocalStructure.myAge;

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Enumerations

We have briefly touched on the concept of enumerations while discussing the layout of the Solidity file earlier in this chapter. Enums are value types comprising a pre-defined list of constant values. They are passed by values and each copy maintains its own value. Enums cannot be declared within functions and are declared within the global namespace of the contract.

Predefined constants are assigned consecutively, increasing integer values starting from zero.

The code illustration shown next declares an `enum` identified as a status consisting of five constant values—`created`, `approved`, `provisioned`, `rejected`, and `deleted`. They have integer values `0`, `1`, `2`, `3`, `4` assigned to them.

A instance of `enum` named `myStatus` is created with an initial value of `provisioned`.

The `returnEnum` function returns the status and it returns the integer value. It is to be noted that `web3` and **Decentralized Applications** (**DApp**) do not understand an `enum` declared within a contract. They will get an integer value corresponding to the `enum` constant.

The `returnEnumInt` function returns an integer value.

The `passByValue` function shows that the `enum` instance maintains its own local copy and does not share with other instances.

The `assignInteger` function shows an example where an integer is assigned as a value to an `enum` instance:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract Enums {

//enum declared
enum status {created, approved, provisioned, rejected, deleted}

//instance of enum with initial value 2
status myStatus = status.provisioned;


    function returnEnum() returns (status)
    {
        status stat = status.created;
        return stat;
    }

   function returnEnumInt() returns (uint)
    {

       status stat = status.approved;
        return uint(stat);
    }

    function passByValue() returns (uint)
    {

       status stat = myStatus;
       myStatus = status.rejected;

        return uint(myStatus);
    }

    function assignInteger() returns (uint)
    {

       status stat = myStatus;

       //casting integer 2 to enum and assigning
       myStatus = status(2);

        return uint(myStatus);
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Address

An address is a 20 bytes data type. It is specifically designed to hold account addresses in Ethereum, which are 160 bits or 20 bytes in size. It can hold contract account addresses as well as externally owned account addresses. Address is a value type and it creates a new copy while being assigned to another variable.

Address has a `balance` property that returns the amount of Ether available with the account and has a few functions for transferring Ether to accounts and invoking contract functions.

It provides the following two functions to transfer Ether:

- `transfer`
- `send`

The `transfer` function is a better alternative for transferring Ether to an account than the `send` function. The `send` function returns a boolean value depending on successful execution of the Ether transfer while the `transfer` function raises an exception and returns the Ether to the caller.

It also provides the following three functions for invoking the `contract` function:

- `Call`
- `DelegateCall`
- `Callcode`

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Mappings

Mappings are one of the most used complex data types in Solidity. Mappings are similar to hash tables or dictionaries in other languages. They help in storing key-value pairs and enable retrieving values based on the supplied key.

Mappings are declared using the `mapping` keyword followed by data types for both key and value separated by the `=>` notation. Mappings have identifiers like any other data type and they can be used to access the mapping.

An example of mapping is as follows:

```
Mapping ( uint => address ) Names ;
```

In the preceding code, the `uint` data type is used for storing the keys and the `address` data type is used for storing the values. `Names` is used as an identifier for the mapping.

Although it is similar to a hash table and dictionary, Solidity does not allow iterating through mapping. A value from mapping can be retrieved if the key is known. The next example illustrates working with mapping. A counter of type `uint` is maintained in a contract that acts as a key and address details are stored and retrieved with the help of functions.

To access any particular value in mapping, the associated key should be used along with the mapping name as shown here:

```
Names[counter]
```

To store a value in mapping, use the following syntax:

```
Names[counter] = <<some value>>
```

Take a look at the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract GeneralMapping {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
        counter = counter + 1;
      Names[counter] = addressDetails;

      return counter; //returns false
    }

   function getMappingMember(uint id) returns (address)
    {
      return Names[id];
    }
}
```

Although mapping doesn't support iteration, there are ways to work round this limitation. The next example illustrates one of the ways to iterate through mapping. Please note that iterating and looping are an expensive operation in Ethereum in terms of gas usage and should generally be avoided. In this example, a separate counter is maintained to keep track of the number of entries stored within the mapping. This counter also acts as the key within the mapping. A local array can be constructed for storing the values from mapping. A loop can be executed using `counter` and can extract and store each value from the mapping into the local array as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract MappingLooping {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
      counter = counter + 1;

      Names[counter] = addressDetails;

      return counter;
    }

   function getMappingMember(uint id) returns (address[])
    {
      address[] memory localBytes = new address[](counter);
      for(uint i=1; i<= counter; i++){
         localBytes[i - 1] = Names[i];
      }

      return localBytes;
    }
}
```

Mapping can only be declared as a state variable whose memory location is of type storage. Mapping cannot be declared within functions as memory mappings. However, mappings can be declared in functions if they refer to mappings declared in state variables, as shown in the following example:

```
Mapping (uint => address) localNames = Names ;
```

This is valid syntax as the `localNames` mapping is referring to the `Names` state variable:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract MappinginMemory {

    mapping (uint => address) Names;

    uint counter;

    function addtoMapping(address addressDetails) returns (uint)
    {
      counter = counter + 1;
      mapping (uint => address) localNames = Names;

      localNames[counter] = addressDetails;

      return counter;
    }

   function getMappingMember(uint id) returns (address)
    {
      return Names[id];
    }
}
```

It is also possible to have nested mapping, that is mapping consisting of mappings. The next example illustrates this. In this example, there is an apparent mapping that maps `uint` to another mapping. The child mapping is stored as a value for the first mapping. The child mapping has the `address` type as the key and the `string` type as value. There is a single mapping identifier and the child or inner mapping can be accessed using this identifier itself as shown in the following code:

```
mapping (uint => mapping(address => string)) accountDetails;
```

To add an entry to this type of nested mapping, the following syntax can be used:

```
accountDetails[counter][addressDetails] = names;
```

Here, `accountDetails` is the mapping identifier and `counter` is the key for parent mapping. The `accountDetails[counter]` mapping identifier retrieves the value from the parent mapping, which in turn happens to be another mapping. Adding the key to the returned value, we can set the value for the inner mapping. Similarly, the value from the inner mapping can be retrieved using the following syntax:

```
accountDetails[counter][addressDetails]
```

Take a look at the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract DemoInnerMapping {

    mapping (uint => mapping(address => string)) accountDetails;
    uint counter;

    function addtoMapping(address addressDetails, bytes name) returns (uint)
    {
        string memory names = string(name);
        counter = counter + 1;
        accountDetails[counter][addressDetails] = names;

        return counter;
    }

   function getMappingMember(address addressDetails) returns (bytes)
    {
        // 0xca35b7d915458ef540ade6068dfe2f44e8fa733c
      return bytes( accountDetails[counter][addressDetails]);
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Summary

This is the first chapter that has explored Solidity in depth. This chapter introduced Solidity, the layout of Solidity files including elements that can be declared at the top level in it. Constructs li pragma, contracts, and elements of contracts were discussed for a layout perspective. A complete immersion into the world of Solidity data types forms the core of this chapter. Value types and reference types were discussed in depth along with types like `int`, `uint`, fixed sized byte arrays, bytes, arrays, strings, structures, enumerations, addresses, boolean, and mappings were discussed in great length along with examples. Solidity provides additional data locations from complex types such as structs and arrays, which were also discussed in depth along with rules that govern their usage.

In the next chapter, we will focus on using some out-of-box variables and functions of smart contracts. Solidity provides numerous global variables and functions to help ease the task of obtaining the current transaction and block context. These variables and function provides contextual information and Solidity code and utilizes them for logic execution. They play a very important role in authoring enterprise-scale smart contracts.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Global Variables and Functions

In Chapter 3, *Introducing Solidity*, you learned about Solidity data types in detail. Data types can be value or reference types. Some reference types such as structs and arrays also have data locations—memory and storage associated with them. Variables could be state variables or variables defined locally within functions. This chapter will focus on variables, their scoping rules, declaration and initialization, conversion rules hoisting, and variables available globally to all contracts. Some global functions will also be discussed in this chapter.

We will cover the following topics in this chapter:

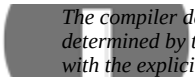
- The `var` data type
- Variable scoping
- Variable conversion
- Variable hoisting
- Block related global variables
- Transaction related global variables
- Mathematical and cryptographic global functions
- Addressing related global variables and functions
- Contract-related global variables and functions

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The var type variables

One Solidity type that was not discussed in the last chapter is the `var` data type. `var` is a special type that can only be declared within a function. There cannot be a state variable in a contract of type `var`. Variables declared with the `var` type are known as **implicitly typed variable** because `var` does not represent any type explicitly. It informs the compiler that its type is dependent and determined by the value assigned to it the first time. Once a type is determined, it cannot be changed.



The compiler decides the final data type for the `var` variables instead of a developer mentioning the type. It is therefore quite possible that the type determined by the `block.difficulty (uint)` current block compiler might not exactly be the type expected by code execution. `var` cannot be used with the explicit usage of memory location. An explicit memory location needs an explicit variable type.

An example of `var` is shown in the following screenshot. Variable `uintVar8` is of type `uint8`, variable `uintVar16` is of type `uint16`, variable `intVar8` is of type `int8` (signed integer), variable `intVar16` is of type `int16` (signed integer), variable `boolVar` is of type bool, variable `stringVar` is of type string, variable `bytesVar` is of type bytes, variable `arrayInteger` is of type `uint8` array, and variable `arrayByte` is of type `bytes10` array:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract VarExamples {

    function VarType()
    {
        var uintVar8 = 10; //uint8
        uintVar8 = 255; //256 is error

        var uintVar16 = 256; //uint16
        uintVar16 = 65535; //aaa = 65536; is error

       var intVar8 = -1; //int8 values -128 to 127

       var intVar16 = -129; //int16 values -32768 to 32767

        var boolVar = true;
        boolVar = false; // 10 is error, 0 is error, 1 is error, -1 is error


        var stringVar = "0x10"; // this is string memory
        stringVar = "10"; // cc =123123123123123123212222222 is error

        var bytesVar = 0x100; // this is byte memory


        var Var = hex"001122FF";

        var arrayInteger = [uint8(1),2];
        arrayInteger[1] = 255;

        var arrayByte = bytes10(0x2222);
        arrayByte = 0x11111111111111111111; //0x111111111111111111111 is error

    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Variables hoisting

Hoisting a concept is where variables need not be declared and initialized before using the variable. The variable declaration can happen at any place within a function, even after using it. This is known as **variable hoisting**. The Solidity compiler extracts all variables declared anywhere within a function and places them at the top or beginning of a function and we all know that declaring a variable in Solidity also initializes them with their respective default values. This ensures that the variables are available throughout the function.

In the following example, `firstVar`, `secondVar`, and `result` are declared towards the end of the function but utilized at the beginning of the function. However, when the compiler generates the bytecode for the contract, it declares all variables as the first set of instructions in a function as shown in the following screenshot:

```
pragma solidity ^0.4.19;

contract variableHoisting {

    function hoistingDemo() returns (uint){

        firstVar = 10;
        secondVar = 20;

        result = firstVar + secondVar;

        uint firstVar;
        uint secondVar;
        uint result;
        return result;


    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Variable scoping

Scoping refers to the availability of a variable within a function and a contract in Solidity. Solidity provides the following two locations where variables can be declared:

- Contract-level global variables—also known as state variables
- Function-level local variables

It is quite easy to understand function-level local variables. They are only available anywhere within a function and not outside.

Contract-level global variables are variables that are available to all functions including constructor, fallback, and modifiers within a contract. Contract-level global variables can have a visibility modifier attached to them. It is important to understand that state data can be viewed across the entire network irrespective of the visibility modifier. The following state variables can only be modified using functions:

- `public`: These state variables are accessible directly from external calls. A `getter` function is implicitly generated by the compiler to read the value of public state variables.
- `internal`: These state variables are not accessible directly from external calls. They are accessible from functions within a current contract and child contracts deriving from it.
- `private`: These state variables are not accessible directly from external calls. They are also not accessible from functions from child contracts. They are only accessible from functions within the current contract.

Let's take a look at the preceding state variables in the following screenshot:

```
pragma solidity ^0.4.19;

contract ScopingDtateVariables {

//   uint64 public myVar = 0;

//   uint64 private myVar = 0;

//  uint64 internal myVar = 0;



}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Type conversion

By now, we know that Solidity is a statically typed language, where variables are defined with specific data types at compile time. The data type cannot be changed for the lifetime of the variable. It means it can only store values that are legal for a data type. For example, uint8 can store values from 0 to 255. It cannot store negative values or values greater than 255. Take a look at the following code to better understand this:

```
pragma solidity ^0.4.19;

contract ErrorDataType {

    function hoistingDemo() returns (uint){

        uint8 someVar = 100;
        someVar = 300; //error

    }

}
```

However, there are times when these conversions are required to copy a value into a variable of one type to another, and these are called **type conversions**. Solidity provides rules for type conversions.

In Solidity, we can perform various kinds of conversion and we will cover these in the following sections.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Implicit conversion

Implicit conversion means that there is no need for an operator, or no external help is required for conversion. These types of conversion are perfectly legal and there is no loss of data or mismatch of values. They are completely type-safe. Solidity allows for implicit conversion from smaller to larger integral types. For example, converting uint8 to uint16 happens implicitly.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Explicit conversion

Explicit conversion is required when a compiler does not perform implicit conversion either because of loss of data or a value containing data not falling within a target data type range. Solidity provides a function for each value type for explicit conversion. Examples of explicit conversion are `uint16` conversion to `uint8`. Data loss is possible in such cases.

The following code listing shows examples for both implicit and explicit conversions:

- `ConvertionExplicitUINT8toUINT256`: This function executed explicit conversion from `uint8` to `uint256`. It is to be noted that this conversion was also possible implicitly.
- `ConvertionExplicitUINT256toUINT8`: This function executed explicit conversion from `uint256` to `uint8`. This conversion will raise a compile time error if the conversion happened implicitly.
- `ConvertionExplicitUINT256toUINT81`: This function shows an interesting aspect of explicit conversion. Explicit conversions are error-prone and should generally be avoided. In this function, an attempt is made to store a large value in a variable of a smaller data type. This results in loss of data and unpredictability. The compiler does not generate an error; however, it tries to fit the value into smaller value and goes in cycle to find a valid value.
- `Conversions`: This function shows an example of implicit and explicit conversions. Some fail and some are legal. In the following screenshot, please read the comments beneath the code to understand them:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity ^0.4.19;

contract ConversionDemo {

    function ConvertionExplicitUINT8toUINT256() returns (uint){
        uint8 myVariable = 10;
        uint256 someVariable = myVariable;
        return someVariable;
    }

    function ConvertionExplicitUINT256toUINT8() returns (uint8){
        uint256 myVariable = 10;
        uint8 someVariable = uint8(myVariable);
        return someVariable;
    }

    function ConvertionExplicitUINT256toUINT81() returns (uint8){
        uint256 myVariable = 10000134;
        uint8 someVariable = uint8(myVariable);
        return someVariable; //returns 6 as return value
    }

    function Convertions() {

        uint256 myVariable = 10000134;
        uint8 someVariable  = 100;
        bytes4 byte4 = 0x65666768;

        // bytes1 byte1 = 0x656667668; //error

        bytes1 byte1 = 0x65;

        //  byte1 = byte4; //error, explicit conversion needed here

        byte1 = bytes1(byte4) ; //explicit conversion

        byte4 = byte1;  //Implicit conversion

        // uint8 someVariable = myVariable; // error, explicit conversion needed here

        myVariable = someVariable; //Implicit conversion

        string memory name = "Ritesh";
        bytes memory nameInBytes = bytes(name); //explicit string to bytes conversion

        name = string(nameInBytes); //explicit bytes to string conversion

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Block and transaction global variables

Solidity provides access to a few global variables that are not declared within contracts but are accessible from code within contracts. Contracts cannot access the ledger directly. A ledger is maintained by miners only; however Solidity provides some information about the current transaction and block to contracts so that they can utilize them. Solidity provides both block-as well as transaction-related variables.

The following code illustratesexamples of using global transaction, block, and message variables:

```
pragma solidity ^0.4.19;

contract TransactionAndMessageVariables {

    event logstring(string);
    event loguint(uint);
    event logbytes(bytes);
    event logaddress(address);
    event logbyte4(bytes4);
    event logblock(bytes32);

    function globalVariable() payable {

       logaddress( block.coinbase ); // 0x94d76e24f818426ae84aa404140e8d5f60e10e7e

       loguint( block.difficulty ); //71762765929000

       loguint( block.gaslimit ); // 6000000

       loguint(  msg.gas ); //2975428

       loguint(  tx.gasprice ); // 1

       loguint(  block.number ); //123

       loguint( block.timestamp ); //1513061946

       loguint( now ); //1513061946

       logbytes( msg.data ); // 0x4048d797

       logbyte4(   msg.sig  ); // // 0x4048d797

       loguint( msg.value ); // 0 or in Wei if ether are send

       logaddress( msg.sender ); //0xca35b7d915458ef540ade6068dfe2f44e8fa733c"

       logaddress( tx.origin ); // 0xca35b7d915458ef540ade6068dfe2f44e8fa733c"

       logblock ( block.blockhash( block.number) ); //0x0000000000000000000000000

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Transaction and message global variables

The following is a list of global variables along with their data types and a description provided as a ready reference:

Variable name	Description
`block.coinbase (address)`	Same as etherbase. Refers to the miner's address.
`block.difficulty (uint)`	Difficulty level of current block.
`block.gaslimit (uint)`	Gas limit for current block.
`block.number (uint)`	Block number in sequence.
`block.timestamp (uint)`	Time when block was created.
`msg.data (bytes)`	Information about the function and its parameters that created the transaction.
`msg.gas (uint)`	Gas unused after execution of transaction.
`msg.sender (address)`	Address of caller who invoked the function.
`msg.sig (bytes4)`	Function identifier using first four bytes after hashing function signature.
`msg.value (uint)`	Amount of wei sent along with transaction.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

`now (uint)`	Current time.
`tx.gasprice (uint)`	The gas price caller is ready to pay for each gas unit.
`tx.origin (address)`	The first caller of the transaction.
`block.blockhash(uint blockNumber) returns (bytes32)`	Hash of the block containing the transaction.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Difference between tx.origin and msg.sender

Careful readers might have noticed in the previous code illustration that both `tx.origin` and `msg.sender` show the same result and output. The `tx.origin` global variable refers to the original external account that started the transaction while `msg.sender` refers to the immediate account (it could be external or another contract account) that invokes the function. The `tx.origin` variable will always refer to the external account while `msg.sender` can be a contract or external account. If there are multiple function invocations on multiple contracts, `tx.origin` will always refer to the account that started the transaction irrespective of the stack of contracts invoked. However, `msg.sender` will refer to the immediate previous account (contract/external) that invokes the next contract. It is recommended to use `msg.sender` over `tx.origin`.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Cryptography global variables

Solidity provides cryptographic functions for hashing values within contract functions. There are two hashing functions—SHA2 and SHA3.

The `sha3` function converts the input into a hash based on the `sha3` algorithm while `sha256` converts the input into a hash based on the `sha2` algorithm. There is another function, `keccak256`, which is an alias of the SHA3 algorithm. It is recommended to use the `keccak256` or `sha3` functions for hashing needs.

The following screenshot of the code segment illustrates this:

```
pragma solidity ^0.4.19;

contract CryptoFunctions {

    function cryptoDemo() returns (bytes32, bytes32, bytes32){

        return (sha256("r"), keccak256("r"), sha3("r"));


    }

}
```

The result of executing this function is shown in the following screenshot. The result of both the `keccak256` and `sha3` functions is the same:

```
{
        "0": "bytes32: 0x454349e422f05297191ead13e21d3db520e5abef52055e4964b82fb213f593a1",
        "1": "bytes32: 0x414f72a4d550cad29f17d9d99a4af64b3776ec5538cd440cef0f03fef2e9e010",
        "2": "bytes32: 0x414f72a4d550cad29f17d9d99a4af64b3776ec5538cd440cef0f03fef2e9e010"
}
```

All three of these functions work on tightly packed arguments, meaning that multiple parameters can be concatenated together to find a hash, as shown in the following code snippet:

```
keccak256(97, 98, 99)
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Address global variables

Every address—externally owned or contract-based, has five global functions and a single global variable. These functions and variables will be explored in depth in subsequent chapters on Solidify functions. The global variable related to the address is called **balance** and it provides the current balance of Ether in wei available at the address.

The functions are as follows:

- `<address>.transfer(uint256 amount)`: This function sends the given amount of wei to `address`, throws on failure
- `<address>.send(uint256 amount) returns (bool)`: This function sends the given amount of wei to `address`, and returns `false` on failure
- `<address>.call(...) returns (bool)`: This function issues a low-level `call`, and returns `false` on failure
- `<address>.callcode(...) returns (bool)`: This function issues a low-level `callcode`, and returns `false` on failure
- `<address>.delegatecall(...) returns (bool)`: This function issues a low-level `delegatecall`, and returns `false` on failure

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Contract global variables

Every contract has the following three global functions:

- `this`: The current contract's type, explicitly convertible to address
- `selfdestruct`: This is an address recipient that destroys the current contract, sending its funds to the given address
- `suicide`: This is an address recipient too alias to `selfdestruct`

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Summary

This chapter, in many ways, was a continuation of previous chapters. Variables were discussed in depth in the first half of this chapter. Variable hoisting, type conversions, details about the `var` data type, and the scope of Solidity variables were elaborated on, along with code examples. The latter half of the chapter focused on globally available variables and functions. Transaction and message related variables, such as `block.coinbase`, `msg.data` and many more, were explained. The difference between `msg.sender` and `tx.origin` along with their usage was also explained in this chapter. This chapter also discussed cryptographic, address, and contract-level functions. However, we will focus on these functions in another chapter later in this book.

The following chapter will focus on Solidity expressions and control structures, covering programming details about loops and conditions. This will be an important chapter because every program needs some kind of looping to perform repetitive tasks and Solidity control structures help implement these. Loops are based on conditions and conditions are written using expressions. These expressions are evaluated and return either `true` or `false`. Stay tuned while we plunge into control structures and expressions in the following chapter.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Expressions and Control Structures

Taking decisions in code is an important aspect of a programming language, and Solidity should also be able to execute different instructions based on circumstances. Solidity provides the `if...else` and `switch` statements for this purpose. It is also important to loop through multiple items and Solidity provides multiple constructs such as `for` loops and `while` statements for this purpose. In this chapter, we will discuss in detail the programming constructs that help you take decisions and loop through a set of values.

This chapter covers the following topics:

- Expressions
- The `if...else` statement
- The `while` statement
- The `for` loop
- The `break` and `continue` keywords
- The `return` statement

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

Solidity expressions

An expression refers to a statement (comprising multiple operands and optionally zero or more operators) that results in a single value, object, or function. The operand can be a literal, variable, function invocation, or another expression itself.

An example of an expression is as follows:

```
Age > 10
```

In the preceding code, `Age` is a variable and `10` is an integer literal. `Age` and `10` are operands and the (`>`) greater than symbol is the operator. This expression returns a single boolean value (`true` or `false`) depending on the value stored in `Age`.

Expressions can be more complex comprising multiple operands and operators, as follows:

```
((Age > 10) && (Age < 20) ) || ((Age > 40) && (Age < 50) )
```

In the preceding code, there are multiple operators in play. The `&&` operator acts as an AND operator between two expressions, which in turn comprises operands and operators. There is also an OR operator represented by the `||` operator between two complex expressions.

Solidity has the following comparison operators that help in writing expressions returning Boolean values:

Operator	Meaning	Sample example
`==`	Equals	`myVar == 10`
`!=`	Not equals	`myVar != 10`
`>`	Greater than	`myVar > 10`
`<`	Less than	`myVar < 10`
`>=`	Greater than or	`myVar >= 10`

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

	equal to	
`<=`	Less than or equal to	`myVar <= 10`

Solidity also provides the following logical operators that help in writing expressions returning Boolean values:

Operator	Meaning	Sample example
`&&`	AND	`( myVar > 10) && (myVar < 10)`
`||`	OR	`myVar != 10`
`!`	NOT	`myVar > 10`

The following operators have precedence in Solidity just like other languages:

Precedence	Description	Operator
1	Postfix increment and decrement	`++, --`
New expression	`new <typename>`	NA
Array subscripting	`<array>[<index>]`	NA
Member access	`<object>.<member>`	NA
Function-like call	`<func>(<args...>)`	NA
Parentheses	`(<statement>)`	NA

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

2	Prefix increment and decrement	`++, --`
Unary plus and minus	`+, -`	NA
Unary operations	`delete`	NA
Logical NOT	`!`	NA
Bitwise NOT	`~`	NA
3	Exponentiation	`**`
4	Multiplication, division, and modulo	`*, /, %`
5	Addition and subtraction	`+, -`
6	Bitwise shift operators	`<<, >>`
7	Bitwise AND	`&`
8	Bitwise XOR	`^`
9	Bitwise OR	`|`
10	Inequality operators	`<, >, <=, >=`
11	Equality operators	`==, !=`
12	Logical AND	`&&`

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

13	Logical OR	`||`
14	Ternary operator	`<conditional> ? <if-true> : <if-false>`
15	Assignment operators	`=, |=, ^=, &=, <<=, >>=, +=, -=, *=, /=, %=`
16	Comma operator	`,`

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The if decision control

Solidity provides conditional code execution with the help of the `if...else` instructions. The general structure of `if...else` is as follows:

```
        if (this condition/expression is true) {
           Execute the instructions here
        }
        else if (this condition/expression is true) {
           Execute the instructions here
        }
        else {
           Execute the instructions here
        }
```

`if and if-else` are keywords in Solidity and they inform the compiler that they contain a decision control condition, for example, `if (a > 10)`. Here, `if` contains a condition that can evaluate to either `true` or `false`. If `a > 10` evaluates to `true` then the code instructions that follow in the pair of double-brackets (`{`) and (`}`) should be executed.

`else` is also a keyword that provides an alternate path if none of the previous conditions are true. It also contains a decision control instruction and executes the code instructions if `a > 10` tends to be `true`.

The following example shows the usage of 'IF'-'ELSE IF' - 'ELSE' conditions. An `enum` with multiple constants is declared. A `StateManagement` function accepts an `uint8` argument, which is converted into an `enum` constant and compared within the `if...else` decision control structure. If the value is `1` then the returned result is `1`; if the argument contains `2` or `3` as value, then the `else...if` portion of code gets executed; and if the value is other than `1`,`2`, or `3` then the else part is executed:

```
pragma solidity ^0.4.19;

contract IfElseExample {

    enum requestState {created, approved, provisioned, rejected, deleted, none}

    function StateManagement(uint8 _state) returns (int result) {

       requestState currentState = requestState(_state);

       if(currentState == requestState(1)){
          result = 1;
       } else if ((currentState == requestState.approved) || (currentState == requestState.provisioned)) {
          result = 2;
       } else {
          currentState == requestState.none;
          result = 3;
       }

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The while loop

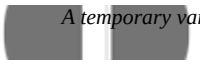
There are times when we need to execute a code segment repeatedly based on a condition. Solidity provides `while` loops precisely for this purpose. The general form of the `while` loop is as follows:

```
Declare and initialize a counter
while (check the value of counter using an expression or condition) {
    Execute the instructions here
    Increment the value of counter
}
```

`while` is a keyword in Solidity and it informs the compiler that it contains a decision control instruction. If this expression evaluates to true then the code instructions that follow in the pair of double-brackets `{` and `}` should be executed. The `while` loop keeps executing until the condition turns false.

In the following example, `mapping` is declared along with `counter`. `counter` helps loop the `mapping` since there is no out-of-the-box support in Solidity to loop `mapping`.

An event is used to get details about transaction information. We will discuss events in detail in the *Events and Logging* section in `Chapter 8`, *Exceptions, Events, and Logging*. For now, it is enough to understand that you are logging information whenever an event is invoked. The `SetNumber` function adds data to mapping and the `getnumbers` function runs a `while` loop to retrieve all entries within the mapping and log them using events.



A temporary variable is used as a counter that is incremented by 1 at every execution of the `while` loop.

The `while` condition checks the value of the temporary variable and compares it with the global counter variable. Based on whether it's true or false, the code within the `while` loop is executed. Within this set of instructions, the value of a counter should be modified so that it can help to exit the loop by making the `while` condition false as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity ^0.4.19;

contract whileLoop {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);
    bytes aa;

    function SetNumber()  {
        blockNumber[counter++] = block.number;
    }

    function getNumbers() {
       uint i = 0;
       while (i < counter) {
           uintNumber( blockNumber[i]  );
           i = i + 1;
       }

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The for loop

One of the most famous and most used loops is the `for` loop, and we can use it in Solidity. The general structure of a `for` loop is as follows:

```
for (initialize loop counter; check and test the counter; increase the value of counter;)

    Execute multiple instructions here
  }
```

`for` is a keyword in Solidity and it informs the compiler that it contains information about looping a set of instructions. It is very similar to the `while` loop; however it is more succinct and readable since all information can be viewed in a single line.

The following code example shows the same solution: looping through a mapping. However, it uses the `for` loop instead of the `while` loop. The `i` variable is initialized, incremented by `1` in every iterator, and checked to see whether it is less than the value of `counter`. The loop will stop as soon as the condition becomes false; that is, the value of `i` is equal to or greater than `counter`:

```
pragma solidity ^0.4.19;


contract ForLoopExample {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;

    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            uintNumber( blockNumber[i]  );
        }

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The do...while loop

The `do...while` loop is very similar to the `while` loop. The general form of a `do...while` loop is as follows:

```
Declare and Initialize a counter
do {
Execute the instructions here
Increment the value of counter
} while(check the value of counter using an expression or condition)
```

There is a subtle difference between the `while` and `do...while` loops. If you notice, the condition in `do...while` is placed towards the end of the loop instructions. The instructions in the `while` loop is not executed at all if the condition is false; however, the instruction in the `do...while` loop get executed once, before the condition is evaluated. So, if you want to execute the instructions at least once, the `do...while` loop should be preferred compared to the `while` loop. Take a look at the following screenshot of a code snippet:

```
pragma solidity ^0.4.19;


contract DowhileLoop {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);
    bytes aa;

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

       uint i = 0;
       do  {
          uintNumber( blockNumber[i]  );
          i = i + 1;
       } while (i < counter);

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The break statement

Loops help iterateing over from the start till it arives on a vector data type. However, there are times when you would like to stop the iteration in between and jump out or exit from the loop without executing the conditional test again. The `break` statement helps us do that. It helps us terminate the loop by passing the control to the first instruction after the loop.

In the following screenshot example, the `for` loop is terminated and control moves out of the `for` loop when the value of `i` is `1` because of the use of the `break` statement. It literally breaks the loop as shown in the following screenshot:

```
pragma solidity ^0.4.19;


contract ForLoopExampleBreak {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            if (i == 1)
                break;
            uintNumber( blockNumber[i]  );

        }

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The continue statement

Loops are based on expressions. The logic of the expression decides the continuity of the loop. However, there are times when you are in between loop execution and would like to go back to the first line of code without executing the rest of the code for the next iteration. The `continue` statement helps us do that.

In the following screenshot, the `for` loop is executed till the end; however the values after `5` are not logged at all:

```
pragma solidity ^0.4.19;

contract ForLoopExampleContinue {

    mapping (uint => uint) blockNumber;
    uint counter;

    event uintNumber(uint);

    function SetNumber()  {

        blockNumber[counter++] = block.number;


    }

    function getNumbers() {

        for (uint i=0; i < counter; i++){
            if ((i > 5) )
               { continue;}
            uintNumber( blockNumber[i]  );

        }

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The return statement

Returning data is an integral part of a Solidity function. Solidity provides two different syntaxes for returning data from a function. In the following code sample, two functions—`getBlockNumber` and `getBlockNumber1`—are defined. The `getBlockNumber` function returns a `uint` without naming the `return` variable. In such cases, developers can resort to using the `return` keyword explicitly to return from the function.

The `getBlockNumber1` function returns `uint` and also provides a name for the variable. In such cases, developers can directly use and return this variable from a function without using the `return` keyword as shown in the following screenshot:

```
pragma solidity ^0.4.19;

contract ReturnValues {

    uint counter;

    function SetNumber()  {

       counter = block.number;


    }

    function getBlockNumber() returns (uint) {

        return counter;

    }

    function getBlockNumber1() returns (uint result) {

        result =  counter;

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Summary

Expressions and control structures are an integral part of any programming language and they are an important element of the Solidity language as well. Solidity provides a rich infrastructure for decision and looping constructs. It provides `if...else` decision control structures and the `for`, `do...while`, and `while` loops for looping over data variables that can be iterated. Solidity also allows us to write conditions and logical, assignment, and other types of statement any that programming language supports.

The following chapter will discuss Solidity and contract functions in detail; these are core elements for writing contracts. Blockchain is about executing and storing transactions and transactions are created when contract functions are executed. Functions can change the state of Ethereum or just return the current state. Functions that change state and those that return—current state will be discussed in detail in the following chapter.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Writing Smart Contracts

Solidity is used to author smart contracts. This chapter is dedicated to smart contracts. It is from here that you will start writing smart contracts. This chapter will discuss the design aspects of writing smart contracts, defining and implementing a contract, and deploying and creating contracts using different mechanisms—using new keywords and known addresses. Solidity provides rich object orientation and this chapter will delve deep into object-oriented concepts and implementations, such as inheritance, multiple inheritance, declaring abstract classes and interfaces, and providing method implementations to abstract functions and interfaces.

This chapter covers the following topics:

- Creating contracts
- Creating contracts via `new`
- Inheritance
- Abstract contracts
- Interfaces

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Smart contracts

What are smart contracts? Everybody bears an expression trying to understand the meaning of contracts and the significance of the word "smart" in reference to contracts. Smart contracts are, essentially, code segments or programs that are deployed and executed in EVM. A contract is a term generally used in the legal world and has little relevance in the programming world. Writing a smart contract in Solidity does not mean writing a legal contract. Moreover, contracts are like any other programming code, containing Solidity code, and are executed when someone invokes them. There is inherently nothing smart about it. A smart contract is a blockchain term; it is a piece of jargon used to refer to programming logic and code that executes within EVM.

A smart contract is very similar to a C++, Java, or C# class. Just as a class is composed of state (variables) and behaviors (methods), contracts contain state variables and functions. The purpose of state variables is to maintain the current state of the contract, and functions are responsible for executing logic and performing update and read operations on the current state.

We have already seen some examples of smart contracts in the previous chapter; however, it's time to dive deeper into the subject.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Writing a simple contract

A contract is declared using the `contract` keyword along with an identifier, as shown in the following code snippet:

```
contract SampleContract {
}
```

Within the brackets comes the declaration of state variables and function definitions. A complete definition of contract was discussed in Chapter 3, *Introducing Solidity*, and I am providing it again for quick reference. This contract has state variables, struct definitions, enum declarations, function definitions, modifiers, and events. State variables, structs, and enums were discussed in detail in Chapter 4, *Global Variables and Functions*. Functions, modifiers, and events will be discussed in detail over the next two chapters. Take a look at the following screenshot of a code snippet depicting contract:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

//contract definition
contract generalStructure {
    //state variables
    int public stateIntVariable; // vriable of integer type
    string stateStringVariable; //variable of string type
    address personIdentifier; // variable of address type
    myStruct human; // variable of structure type
    bool constant hasIncome = true; //variable of constant nature

    //structure definition
    struct myStruct {
        string name; //variable fo type string
        uint myAge; // variable of unsigned integer type
        bool isMarried; // variable of boolean type
        uint[] bankAccountsNumbers; // variable - dynamic array of unsigned integer
    }

    //modifier declaration
    modifier onlyBy(){
        if (msg.sender == personIdentifier) {
            _;
        }
    }

    // event declaration
    event ageRead(address, int );

    //enumeration declaration
    enum gender {male, female}

    //function definition
    function getAge (address _personIdentifier) onlyBy() payable external returns (uint) {

       human =  myStruct("Ritesh",10,true,new uint[](3)); //using struct myStruct

       gender _gender = gender.male; //using enum

       ageRead(personIdentifier, stateIntVariable);
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Creating contracts

There are the following two ways of creating and using a contract in Solidity:

- Using the `new` keyword
- Using the address of the already deployed contract

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Using the new keyword

The `new` keyword in Solidity deploys and creates a new contract instance. It initializes the contract instance by deploying the contract, initializing the state variables, running its constructor, setting the `nonce` value to one, and, eventually, returns the address of the instance to the caller. Deploying a contract involves checking whether the requestor has provided enough gas to complete deployment, generating a new account/address for contract deployment using the requestor's `address` and `nonce` value, and passing on any Ether sent along with it.

In the next screenshot, two contracts, `HelloWorld` and `client`, are defined. In this scenario, one contract (`client`) deploys and creates a new instance of another contract (`HelloWorld`). It does so using the `new` keyword as shown in the following code snippet:

```
HelloWorld myObj = new HelloWorld();
```

Let's take a look at the following screenshot:

```
pragma solidity 0.4.19;

contract HelloWorld {
    uint private simpleInt;

    function   getValue() public view returns (uint) {
        return simpleInt;
    }

    function  setValue(uint _value) public {
        simpleInt =  _value;
    }
}


contract client  {

    function useNewKeyword() public  returns (uint) {

       HelloWorld myObj = new HelloWorld();

       myObj.setValue(10);

       return myObj.getValue();

    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Using address of a contract

This method of creating a contract instance is used when a contract is already deployed and instantiated. This method of creating a contract uses the address of an existing, deployed contract. No new instance is created; rather, an existing instance is reused. A reference to the existing contract is made using its address.

In the next code illustration, two contracts, `HelloWorld` and `client`, are defined. In this scenario, one contract(`client`) uses an already known address of another contract to create a reference to it (`HelloWorld`). It does so using the `address` data type and casting the actual address to the `HelloWorld` contract type. The `myObj` object contains the address of an existing contract, as shown in the following code snippet:

```
HelloWorld myObj = HelloWorld(obj);
```

Let's take a look at the following screenshot:

```
pragma solidity 0.4.19;

contract HelloWorld {

    uint private simpleInt;

    function   GetValue() public view returns (uint) {
        return simpleInt;
    }

    function  SetValue(uint _value) public {
        simpleInt =  _value;
    }
}


contract client  {

    address obj ;

    function setObject(address _obj) external   {
        obj = _obj;
    }

    function UseExistingAddress() public  returns (uint) {
       HelloWorld myObj = HelloWorld(obj);
       myObj.SetValue(10);
       return myObj.GetValue();
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Constructors

Solidity supports declaring a constructor within a contract. Constructors are optional in Solidity and the compiler induces a default constructor when no constructor is explicitly defined. The constructor is executed once while deploying the contract. This is quite different from other programming languages. In other programming languages, a constructor is executed whenever a new object instance is created. However, in Solidity, a constructor is executed are deployed on EVM. Constructors should be used for initializing state variables and, generally, writing extensive Solidity code should be avoided. The constructor code is the first set of code that is executed for a contract. There can be at most one constructor in a contract, unlike constructors in other programming languages. Constructors can take parameters and arguments should be supplied while deploying the contract.

A constructor has the same name as that of the contract. Both the names should be the same. A constructor can be either `public` or `internal`, from a visibility point of view. It cannot be `external` or `private`. A constructor does not return any data explicitly. In the following example, a constructor with the same name as that of the `HelloWorld` contract is defined. It sets the storage variable value to `5`, as shown in the following screenshot:

```
pragma solidity 0.4.19;

contract HelloWorld {

    uint private simpleInt;

    function HelloWorld() public {
        simpleInt = 5;
    }

    function   GetValue() public view returns (uint) {
        return simpleInt;
    }

    function  SetValue(uint _value) public {
        simpleInt =  _value;
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Contract composition

Solidity supports contract composition. Composition refers to combining multiple contracts or data types together to create complex data structures and contracts. We have already seen numerous examples of contract composition before. Refer to the code snippet for creating contracts using the `new` keyword shown earlier in this chapter. In this example, the `client` contract is composed of the `HelloWorld` contract. Here, `HelloWorld` is an independent contract and `client` is a dependent contract. `client` is a dependent contract because it is dependent on the `HelloWorld` contract for its completeness. It is a good practice to break down problems into multi-contract solutions and compose them together using contract composition.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Inheritance

Inheritance is one of the pillars of object orientation and Solidity supports inheritance between smart contracts. Inheritance is the process of defining multiple contracts that are related to each other through parent-child relationships. The contract that is inherited is called the **parent contract** and the contract that inherits is called the **child contract**. Similarly, the contract has a parent known as the **derived class** and the parent contract is known as a **base contract**. Inheritance is mostly about code-reusability. There is a is-a relationship between base and derived contracts and all public and internal scoped functions and state variables are available to derived contracts. In fact, Solidity compiler copies the base contract bytecode into derived contract bytecode. The `is` keyword is used to inherit the base contract in the derived contract.

It is one of the most important concepts that should be mastered by every Solidity developer because of the way contracts are versioned and deployed.

Solidity supports multiple types of inheritance, including multiple inheritance.

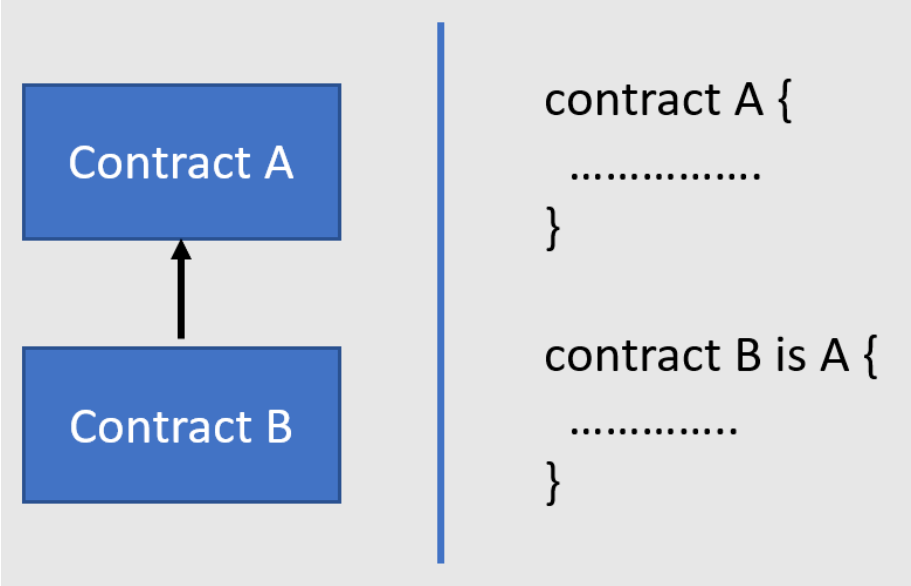
Solidity copies the base contracts into the derived contract and a single contract is created with inheritance. A single address is generated that is shared between contracts in a parent-child relationship.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Single inheritance

Single inheritance helps in inheriting the variables, functions, modifiers, and events of base contracts into the derived class. Take a look at the following diagram:



The next code snippets help to explain single inheritance. You will observe that there are two contracts, `ParentContract` and `ChildContract`. The `ChildContract` contract inherits from `ParentContract`. `ChildContract` will inherit all public and internal variables and functions. Anybody using `ChildContract`, as seen in the `client` contract, can invoke both `GetInteger` and `SetInteger` functions as if they were defined in `ChildContract`, as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract ParentContract {
    uint internal simpleInteger;

    function SetInteger(uint _value) external {
        simpleInteger =  _value;
    }
}


contract ChildContract is ParentContract  {
    bool private simpleBool;

    function   GetInteger() public view returns (uint) {
        return simpleInteger;
    }

}



contract Client  {
    ChildContract pc = new ChildContract();

    function workWithInheritance() public  returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }

}
```

All functions in Solidity contracts are virtual and are based on contract instance. An appropriate function—either in the base or derived class is invoked. This topic is known as **polymorphism** and is covered in a later section in this chapter.

The order of invocation of the contract constructor is from the base most contract to the derive most contract.

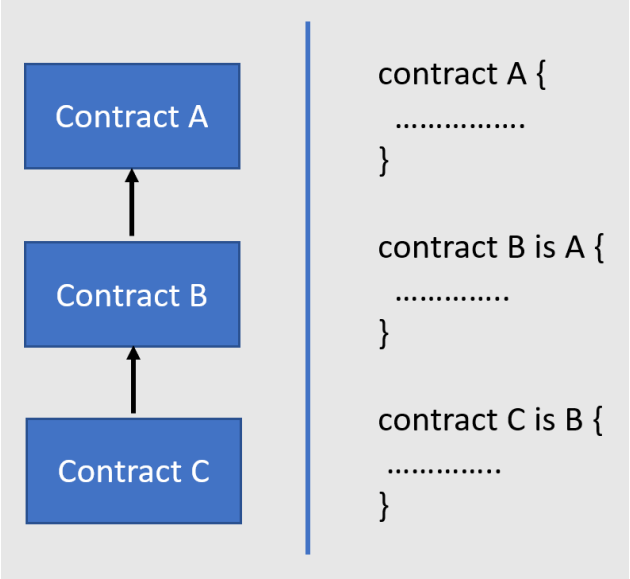
Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Multi-level inheritance

Multi-level inheritance is very similar to single inheritance; however, instead of just a single parent-child relationship, there are multiple levels of parent-child relationship.

This is shown in the following diagram. **Contract A** is the parent of **Contract B** and **Contract B** is the parent of **Contract C**:

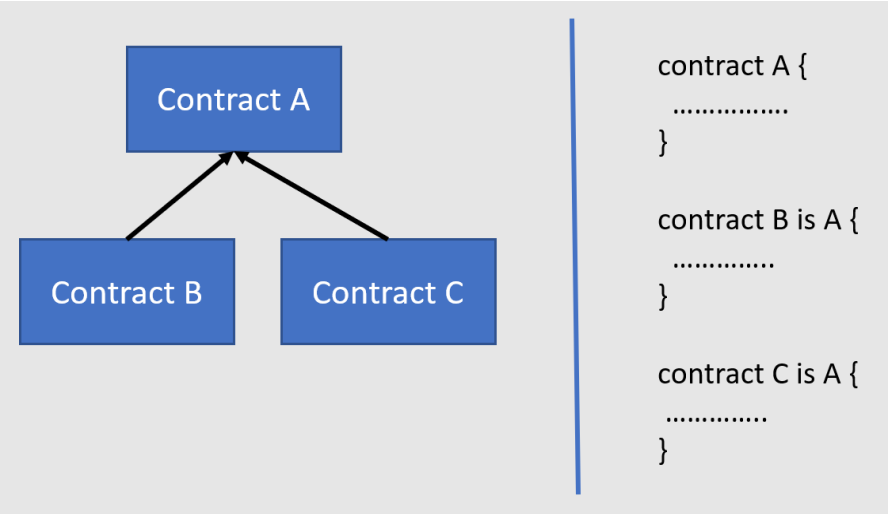


Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Hierarchical inheritance

Hierarchical inheritance is again similar to simple inheritance. Here, however, a single contract acts as a base contract for multiple derived contracts. This is shown in the following diagram. Here, **Contract A** is derived in both **Contract B** and **Contract C**:

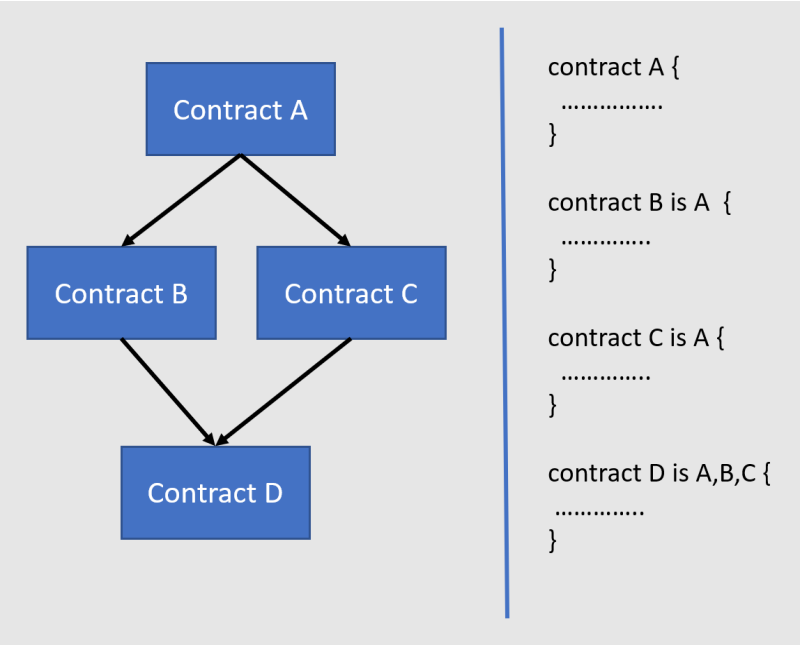


Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Multiple inheritance

Solidity supports multiple inheritance. There can be multiple levels of single inheritance. However, there can also be multiple contracts that derive from the same base contract. These derived contracts can be used as base contracts together in further child classes. When contracts inherit from such child contracts together, there is multiple inheritance, as shown in the following diagram:



The next screenshot shows an example of multiple inheritance. In this example, `SumContract` acts as a base contract that is derived into the `MultiContract` and `DivideContract` contracts. The `SumContract` contract provides an implementation for the `Sum` function and the `MultiContract` and `DivideContract` contracts provide an implementation of the `Multiply` and `Divide` functions, respectively. Both `MultiContract` and `DivideContract` are inherited in `SubContract`. The `SubContract` contract provides an implementation of the `Sub` function. The `client` contract is not a part of the parent-child hierarchy and is consuming other contracts. The `client` contract creates an instance of `SubContract` and calls the `Sum` method on it.

Solidity follows the path of Python and uses **C3 Linearization**, also known as **Method Resolution Order** (**MRO**), to force a specific order in graphs of base contracts. The contracts should follow a specific order while inheriting, starting from the base contract through to the most derived contract. An example of such sequencing is shown next, in which the `SubContract` contract is derived from `SumContract`, `DivideContract`, and `MultiContract`.

The following screenshot of the code example shows that `MultiContract` is an immediate parent contract for the `SubContract` contract, followed by `DivideContract` and `SumContract`:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract SumContract {

    function Sum(uint a, uint b) public returns (uint) {
        return  a + b;
    }

}

contract MultiContract is SumContract  {

    function Multiply(uint a, uint b) public returns (uint) {
        return  a * b;
    }

}

contract DivideContract is SumContract  {

    function Divide(uint a, uint b) public returns (uint) {
        return  a / b;
    }
}

contract SubContract is SumContract, MultiContract,DivideContract

    function sub(uint a, uint b) public returns (uint) {
        return  a - b;
    }
}

contract client  {

    function workWithInheritance() public  returns (uint) {
        uint a = 20;
        uint b = 10;
        SubContract subt = new SubContract();
        return subt.Sum(a,b);
    }
}
```

It is also possible to invoke a function specific to a contract by using the contract name along with the function name.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Encapsulation

Encapsulation is one of the most important pillars of OOP. Encapsulation refers to the process of hiding or allowing access to state variables directly for changing their state. It refers to the pattern of declaring variables that cannot be accessed directly by clients and can only be modified using functions. This helps in constraint access to variables but, at the same time, allows enough access to class for taking action on it. Solidity provides multiple visibility modifiers such as `external`, `public`, `internal`, and `private` that affects the visibility of state variables within the contract in which they are defined, inheriting child contracts or outside contracts.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Polymorphism

Polymorphism means having multiple forms. There are the following two types of polymorphism:

- Function polymorphism
- Contract polymorphism

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Function polymorphism

Function polymorphism refers to declaring multiple functions within the same contract or inheriting contracts having the same name. The functions differ in the parameter data types or in the number of parameters. Return types are not taken into consideration for determining valid function signatures for polymorphism. This is also known as **method overloading**.

The next code segment illustrates a contract that contains two functions, which have the same name but different data types for incoming parameters. The first function, `getVariableData`, accepts `int8` as its parameter data type, while the next function having the same name accepts `int16` as its parameter data type. It is absolutely legal to have the same function name with a different number of parameters of different data types for incoming parameters as shown in the following screenshot:

```
pragma solidity ^0.4.19;

contract HelloFunctionPolymorphism
{
    function getVariableData(int8 data) public constant returns(int8 output)
    {
        return data;
    }

    function overloadedFunction(int16 data) public constant returns(int16 output)
    {
        return data;
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Contract polymorphism

Contract polymorphism refers to using multiple contract instances interchangeably when the contracts are related to each other by way of inheritance. Contract polymorphism helps in invoking derived contract functions using a base contract instance.

Let's understand this concept with the help of code listing shown next.

A parent contract contains two functions, `SetInteger` and `GetInteger`. A child contract inherits from a parent contract and provides its own implementation of `GetInteger`. The child contract can be created using the `ChildContract` variable data type and it can also be created using the parent contract data type. Polymorphism allows the use of any contract in a parent-child relationship with the base type contract variable. The contract instance decides which function will be invoked—the base or derived contract.

Take a look at the following code snippet:

```
ParentContract pc = new ChildContract();
```

The preceding code creates a child contract and stores the reference in the parent contract type variable. This is how contract polymorphism is implemented in Solidity as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity ^0.4.19;

contract ParentContract {

    uint internal simpleInteger;

    function SetInteger(uint _value) public {
        simpleInteger =  _value;
    }

    function GetInteger() public view returns (uint) {
        return 10;
    }
}

contract ChildContract is ParentContract  {

    function GetInteger() public view returns (uint) {
        return simpleInteger;
    }
}

contract client  {

    ParentContract pc = new ChildContract();

    function workWithInheritance() public returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Method overriding

Method overriding refers to redefining a function available in the parent contract having the same name and signature in the derived contract. The next code segment shows this. A parent contract contains two functions, `SetInteger` and `GetInteger`. A child contract inherits from the parent contract and provides its own implementation of `GetInteger` by overriding the function.

Now, when a call to the `GetInteger` function is made on the child contract even while using a parent contract variable, the child contract instance function is invoked. This is because all functions in contracts are virtual and based on contract instance; the most derived function is invoked, as shown in the following screenshot:

```
pragma solidity ^0.4.19;

contract ParentContract {

    uint internal simpleInteger;

    function SetInteger(uint _value) public {
        simpleInteger =  _value;
    }

    function GetInteger() public view returns (uint) {
        return 10;
    }
}

contract ChildContract is ParentContract  {

    function GetInteger() public view returns (uint) {
        return simpleInteger;
    }
}

contract client  {

    ParentContract pc = new ChildContract();

    function workWithInheritance() public returns (uint) {
        pc.SetInteger(100);
        return pc.GetInteger();
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Abstract contracts

Abstracts contracts are contracts that have partial function definitions. You cannot create an instance of an abstract contract. An abstract contract must be inherited by a child contract for utilizing its functions. Abstract contracts help in defining the structure of a contract and any class inheriting from it must ensure to provide an implementation for them. If the child contract does not provide the implementation for incomplete functions, even its instance cannot be created. The function signatures terminate using the semicolon, `;`, character. There is no Solidity-provided keyword to mark a contract as abstract. A contract becomes an abstract class if it has functions without implementation.

The screenshot shown next is an implementation of an abstract contract. The `abstractHelloWorld` contract is an abstract contract as it contains a couple of functions without any definitions. `GetValue` and `SetValue` are function signatures without any implementation. There is another method that returns a constant. The purpose of `AddaNumber` is to show that there can be functions within an abstract contract containing implementation as well. The `abstractHelloWorld` abstract contract is inherited by the `HelloWorld` contract that provides implementation for all the methods. The `client` contract creates an instance of the `HelloWorld` contract using the base contract variable and invokes its functions as shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity 0.4.19;

contract abstractHelloWorld {
    function  GetValue() public view returns (uint);
    function  SetValue(uint _value) public;

    function AddaNumber(uint _value) public returns (uint){
        return 10;
    }

}


contract HelloWorld is abstractHelloWorld{
    uint private simpleInteger;

    function   GetValue() public view returns (uint) {
        return simpleInteger;
    }

    function  SetValue(uint _value) public {
        simpleInteger =  _value;

    }

    function AddaNumber(uint _value) public returns (uint){
        return simpleInteger + _value;
    }
}


contract client  {

    abstractHelloWorld myObj ;

    function client(){
        myObj = new HelloWorld();
    }

    function   GetSetIntegerValue() public  returns (uint) {
        myObj.SetValue(100);
       return  myObj.AddaNumber(200);
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Interfaces

Interfaces are like abstract contracts, but there are differences. Interfaces cannot contain any definition. They can only contain function declarations. It means functions in interfaces cannot contain any code. They are also known as **pure** abstract contracts. An interface can contain only the signature of functions. It also cannot contain any state variables. They cannot inherit from other contracts or contain enums or structures. However, interfaces can inherit other interfaces. The function signatures terminate using the semicolon `;` character. Interfaces are declared using the `interface` keyword following by an identifier. The next code example shows an implementation of the interface. Solidity provides the `interface` keyword for declaring interfaces. The `IHelloWorld` interface is defined containing two function signatures—`GetValue` and `SetValue`. There are no functions containing any implementation. `IHelloWorld` is implemented by the `HelloWorld` contract. Contract intent to use this contract would create an instance as it would do normally as shown in the following screenshot:

```
pragma solidity 0.4.19;

interface IHelloWorld {
    function  GetValue() public view returns (uint);
    function  SetValue(uint _value) public;
}


contract HelloWorld is IHelloWorld{
    uint private simpleInteger;

    function   GetValue() public view returns (uint) {
        return simpleInteger;
    }

    function  SetValue(uint _value) public {
        simpleInteger =  _value;

    }
}


contract client  {

    IHelloWorld myObj ;

    function client(){
        myObj = new HelloWorld();
    }

    function   GetSetIntegerValue() public  returns (uint) {
        myObj.SetValue(100);
       return  myObj.GetValue();
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Summary

This brings us to the end of this chapter. It was a heavy chapter that focused primarily on smart contracts, the different ways to create an instance, and all the important object-oriented concepts related to them, including inheritance, polymorphism, abstraction, and encapsulation. Multiple types of inheritance can be implemented in Solidity. Simple, multiple, hierarchical, and multi-level inheritance were discussed, along with usage and implementation of abstract contracts and interfaces. It should be noted that using inheritance in Solidity, there is eventually just one contract that is deployed instead of multiple contracts. There is just one address that can be used by any contract with a parent-child hierarchy.

The next chapter will focus purely on functions within contracts. Functions are central to writing effective Solidity contracts. These are functions that help change the contract state and retrieve them. Without functions, having any meaningful smart contracts is difficult. Functions have different visibility scope, multiple attributes are available that affect their behavior, and also help in accepting Ether. Stay tuned for a function ride in the next chapter!

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Functions, Modifiers, and Fallbacks

Solidity is maturing and providing advanced programming constructs so that users can write better smart contracts. This chapter is dedicated to some of the most important smart contract constructs, such as functions, modifiers, and fallbacks. Functions are the most important element of a smart contract after state variables. It is functions that help to create transactions and implement custom logic in Ethereum. There are various types of functions, which will be discussed in depth in this chapter. Modifiers are special functions that help in writing more readily available and modular smart contracts. Fallbacks are a concept unique to contract-based programming languages, and they are executed when a function call does not match any existing declared method in the contract. Finally, every function has visibility attached to it that affects its availability to the external caller, other contracts, and contracts in inheritance.

This chapter covers the following topics:

- Input parameters and output parameters
- Returning multiple parameters
- View functions
- Pure functions
- Scopes and declarations
- Visibility and getters
- Internal function calls
- External function calls
- Modifiers
- Fallback functions

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Function input and output

Functions would not be that interesting if they didn't accept parameters and return values. Functions are made generic with the use of parameters and return values. Parameters can help in changing function execution and providing different execution paths. Solidity allows you to accept multiple parameters within the same function; the only condition is that their identifiers should be uniquely named.

The following code snippets show the following multiple functions, each with different constructs for parameters and return values:

1. The first function, `singleIncomingParameter`, accepts one parameter named `_data` of type `int` and returns a single return value that is identified using `_output` of type `int`. The `function` signature provides constructs to define both the incoming parameters and return values. The `return` keyword in the `function` signature helps define the return types from the function. In the following code snippet, the `return` keyword within the function code automatically maps to the first `return` type declared in the `function` signature:

```
function singleIncomingParameter(int _data) returns (int
_output) {
    return _data * 2;
}
```

1. The second function, `multipleIncomingParameter`, accepts two parameters: `_data` and `_data2`, which are both of type `int` and return a single return value identified using `_output` of type `int`, as follows:

```
function multipleIncomingParameter(int _data, int _data2)
returns (int _output) {
    return _data * _data2;
}
```

1. The third function, `multipleOutgoingParameter`, accepts one parameter, `_data`, of type `int` and returns two return values identified using `square` and `half`, which are both of type `int`. In the following code snippet, returning multiple parameters is something unique to Solidity and is not found in many programming languages:

```
function multipleOutgoingParameter(int _data) returns (int
square, int half)
{
    square = _data * _data;
    half = _data /2 ;
}
```

1. The fourth function, `multipleOutgoingTuple`, is similar to the third function mentioned previously. However, instead of assigning return values as separate statements and variables, it returns values as a tuple. A **tuple** is a custom data structure consisting of multiple variables, as shown in the following code snippet:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
function multipleOutgoingTuple(int _data) returns (int square,
int half)
 {
     (square, half) = (_data * _data,_data /2 );
 }
```

The entire contract code is shown in the following screenshot:

```
pragma solidity ^0.4.19;


contract Parameters {


    function singleIncomingParameter(int _data) returns (int _output) {
        return _data * 2;
    }

    function multipleIncomingParameter(int _data, int _data2) returns (int _output) {
        return _data * _data2;
    }

    function multipleOutgoingParameter(int _data) returns (int square, int half) {
        square = _data * _data;
        half = _data /2 ;
    }

    function multipleOutgoingTuple(int _data) returns (int square, int half) {
        (square, half) = (_data * _data,_data /2 );
    }


}
```

It is also possible to declare parameters without any identifier at all. This feature does not have much utility, however, as those parameters cannot be referenced within the function code. Similarly, return values can be declared without any name.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Modifiers

Modifiers are another concept unique to Solidity. Modifiers help in modifying the behavior of a function. Let's try to understand this with the help of an example. The following code does not use modifiers; in this contract, two state variables, two functions, and a constructor are defined. One of the state variables stores the address of the account deploying the contract. Within the constructor, the global variable `msg.sender` is used to input the account value in the owner state variable. The two functions check whether the caller is the same as the account that deployed the contract; if it is, the function code is executed, otherwise it ignores the rest of the code. While this code works as is, it can be made better both in terms of readability and manageability. This is where modifiers can help. In this example, the checks are made using the `if` conditional statements. Later, in the next chapter, we will see how to use new Solidity constructs, such as `require` and `assert`, to execute the same checks without `if` conditions. Take a look at the following screenshot of the code snippet depicting modifiers:

```
pragma solidity ^0.4.17;


contract ContractWithoutModifier {

    address owner;
    int public mydata;

    function ContractWithoutModifier(){
        owner = msg.sender;
    }

    function AssignDoubleValue(int _data) public  {
        if(msg.sender == owner) {
            mydata = _data * 2;
        }
    }

    function AssignTenerValue(int _data) public  {
        if(msg.sender == owner) {
            mydata = _data * 10;
        }
    }

}
```

Modifiers are special functions that change the behavior of a function. Here, the function code remains the same, but the execution path of a function changes. Modifiers can only be applied to functions. Let's now see how to write the same contract using modifiers shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity ^0.4.17;


contract ContractWithModifier {

    address owner;
    int public mydata;

    function ContractWithoutModifier(){
        owner = msg.sender;
    }

    modifier isOwner {
        // require(msg.sender == owner);
        if(msg.sender == owner) {
            _;
        }
    }

    function AssignDoubleValue(int _data) public isOwner {
            mydata = _data * 2;
    }

    function AssignTenerValue(int _data) public  {
            mydata = _data * 10;
    }

}
```

The contract shown here has the same constructs: a constructor, two state variables, and two functions. It also has an additional special function that is defined using the `modifier` keyword. The function code for both the `AssignDoubleValue` and `AssignTenerValue` functions are different, although they have similar functionality. These functions do not use the `if` condition to check whether the caller of the function is the same as the account that deployed the contract; instead, these functions are decorated with the modifier name in their signature.

Let's now try to understand the modifier construct in Solidity and its usage.

Modifiers are defined using the `modifier` keyword and an identifier. The code for modifier is placed within curly brackets. The code within a modifier can validate the incoming value and can conditionally execute the called function after evaluation. The _ identifier is of special importance here—its purpose is to replace itself with the function code that is invoked by the caller.

When a caller calls the `AssignDoubleValue` function, which is decorated with the `isOwner` modifier, the modifier takes control of the execution and replaces the _ identifier with the called function code, that is, `AssignDoubleValue`. Eventually, in EVM, the modifier looks like the following code during runtime:

```
modifier isOwner {
// require(msg.sender == owner);
if(msg.sender == owner) {
mydata = _data * 2;
}
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The same modifier can be applied to multiple functions, and the _ identifier can be replaced to the called function code.

This helps in writing cleaner, more readable, and more maintainable code. Developers do not have to keep repeating the same code in every function or check for the incoming value when executing a function.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The view, constant, and pure functions

Solidity provides special modifiers for functions, such as `view`, `pure`, and `constant`. These are also known as **state mutability** attributes because they define the scope of changes allowed within the Ethereum global state. The purpose of these modifiers is similar to those discussed previously, but there are some small differences. This section will detail the use of these keywords.

Writing smart contract functions helps primarily with the following three activities:

- Updating state variables
- Reading state variables
- Logic execution

The execution of functions and transactions costs gas and is not free of cost. Every transaction needs a specified amount of gas based on its execution and callers are responsible for supplying that gas for successful execution. This is true for transactions or for any activity that modifies the global state of Ethereum.

There are functions that are only responsible for reading and returning the state variable, and these are like property getters in other programming languages. They read the current value in a state variable and return values back to the caller. These functions do not change the state of Ethereum. Ethereum's documentation (`http://solidity.readthedocs.io/en/v0.4.21/contracts.html`) mentions the following statements in relation to things that modify state:

- Writing to state variables
- Emitting events
- Creating other contracts
- Using `selfdestruct`
- Sending Ether via calls
- Calling any function not marked `view` or `pure`
- Using low-level calls
- Using inline assembly that contains certain opcodes

Solidity developers can mark their functions with the `view` modifier to suggest to EVM that this function does not change the Ethereum state or any activity mentioned before. Currently, this is not enforced, but it is expected to be in the future.

An example of the `view` function is shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
pragma solidity ^0.4.17;


contract ViewFunction {

    function GetTenerValue(int _data) public view returns (int)  {
            return _data * 10;
    }

}
```

If you have functions that just return values without any modification of state, they can be marked with the `view` function.

It is also worth noting that the `view` functions are also known as **constant** functions. The `constant` functions were used in previous versions of Solidity.

The `pure` functions are more restrictive in terms of state mutability when compared to the `view` functions; however, their purpose is the same, that is, to restrict state mutability. It is also worth noting that even the `pure` functions are not enforced as of the time of writing, but we expect it to be in the future.

The `pure` functions add further restrictions on top of the `view` functions; for example, a `pure` function is not allowed to even read the current state of Ethereum. In short, the `pure` functions disallow reading and writing to Ethereum's global state. The additional activities not allowed according to documentation include the following:

- Reading from state variables
- Accessing `this.balance` or `<address>.balance`
- Accessing any of the members of `block`, `tx`, and `msg` (with the exception of `msg.sig` and `msg.data`)
- Calling any function not marked `pure`
- Using inline assembly that contains certain opcodes

The previous function has been rewritten as a `pure` function in the following screenshot:

```
pragma solidity ^0.4.17;


contract PureFunction {

    function GetTenerValue(int _data) public pure returns (int)  {
            return _data * 10;
    }

}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The address functions

In the chapter relating to data types, we purposely did not explain the functions related to the `address` data type. Although these functions could have been covered there, some of these functions can execute a fallback function automatically, and hence it is covered here.

Address provides five functions and a single property.

The only property provided by `address` is the `balance` property, which provides the balance available in an account (contract or individual) in wei, as shown in the following code snippet:

```
<<account>>.balance ;
```

In the preceding code, `account` is a valid Ethereum address and this returns the balance available in this in terms of wei.

Now, let's take a look at the methods provided by an account.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The send method

The `send` method is used to send Ether to a contract or to an individually owned account. Take a look at the following code depicting the `send` method:

```
<<account>>.send(amount);
```

The `send` function provides 2,300 gas as a fixed limit, which cannot be superseded. This is especially important when sending an amount to a contract address. To send an amount to an individually owned account, this amount of gas is enough. The `send` function returns a boolean `true`/`false` as a return value. In this case, an exception is not returned; instead, `false` is returned from the function. If everything goes right in an execution, `true` is returned from the function. If `send` is used along with the contract address, it will invoke the fallback function on the contract. We will investigate fallback functions in detail in the following section.

Now, let's see an example of the `send` function, as shown in the following screenshot:

```
function SimpleSendToAccount() public returns (bool) {
    return msg.sender.send(1);
}
```

In the preceding screenshot, the `send` function sent `1` wei to the caller of the `SimpleSendToAccount` function. We already learned about `msg.sender` in previous chapters dealing with global variables.

`send` is a low-level function and should be used with caution as it can invoke fallback functions that may recursively call back within the calling contract again and again. There is a pattern known as **Check-Deduct-Transfer** (**CDF**), or sometimes as **Check-Effects-Interaction** (**CEI**), which we look at in the following screenshot. In this pattern, it is assumed that balances are maintained within a mapping. The `mapping` consists of an address and its associated balance, as shown in the following screenshot:

```
mapping (address => uint) balance;

function SimpleSendToAccount(uint amount) public returns (bool) {
    if(balance[msg.sender] >= amount ) {
        balance[msg.sender] -= amount;
        if (msg.sender.send(amount) == true) {
            return true;
        }
        else {
            balance[msg.sender] += amount;
            return false;
        }
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

In this example, a check is first made to see if the caller has a sufficient balance to withdraw funds. If it has, we can reduce the amount from the existing balance and call the `send` method. Then, we must check that `send` is successful; if not, return the amount.

It is worth noting that a lot of sources claim `send` is being deprecated, but I do not think it is. There are specific usages of the `send` function still available, such as sending an amount to multiple accounts. However, a new function transfer has been introduced to send Ether from one account to another; an even better solution would be to ask other contracts and accounts to call a specific method to withdraw the amount.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The transfer method

The `transfer` method is similar to the `send` method. It is responsible for sending Ether or wei to an address. However, the difference here is that `transfer` raises an exception in the case of execution failure, instead of returning `false` , and all changes are reverted. Take a look at the `transfer` method in the following screenshot:

```
function SimpleTransferToAccount() public  {
    msg.sender.transfer(1);
}
```

The `transfer` method is preferred over the `send` method as it raises an exception in the event of an error, meaning exceptions are bubbled up in the stack and halt execution.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

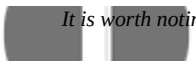
EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The call method

The `call` method has resulted in a lot of confusion among developers. There is a `call` method available via the `web3.eth` object, and there is also the `<<address>>.call` function. These are two different functions that have different purposes.

The `web3.eth` call method can only make calls to a node it is connected to and is a read- only operation. It is not allowed to change the state of Ethereum. It does not generate a transaction nor does it consume any gas. It is used to call the `pure`, `constant`, and `view` functions.

On the other hand, call function provided by address data type can call any function available within a contract. There are times when the interface of contract, more commonly known as ABI, is not available, and so the only way to invoke a function is to use the `call` method. This method does not adhere to ABI and can call any function on a need-to-know basis. There is no compile time check available for these calls, and they return a boolean value of either `true` or `false`.



It is worth noting that it is not an ideal practice to call a `contract` function using the `call` method, as there are no checks and validation involved.

Every function in a contract is identified at runtime using a 4-bytes identifier. This 4-bytes identifier is the trimmed-down hash of a function name along with its parameter types. After hashing the function name and parameter types, the first four bytes are considered as the function identifier. The `call` function accepts these bytes to call the function as the first parameter and the actual parameter values as subsequent parameters.

A `call` function without any function parameter is shown in the following code. Here, `SetBalance` does not take any parameter:

```
myaddr.call(bytes4(sha3("SetBalance()")));
```

A `call` function with a function parameter is shown in the following snippet. Here, `SetBalance` takes a single `uint` parameter:

```
myaddr.call(bytes4(sha3("SetBalance(uint256)")), 10);
```

It is also worth noting that the `send` function seen previously actually calls the `call` function internally by supplying zero gas to the function.

The following code example shows all the possible ways of using this function. In this example, a contract named `EtherBox` is created with the following two simple functions:

- `SetBalance`: It has a single state variable, and the purpose of this function is to add `10` in every invocation to the existing value of the state variable

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

- `GetBalance`: This function is responsible for returning the current value of a state variable

Another contract named `usingCall` is created to invoke methods on the `EtherBox` contract via the `call` function. Let's take a look at the following functions mentioned in the upcoming code example:

1. `SimpleCall`: This function creates an instance of the `EtherBox` contract and converts it into an address. Using this address, the `call` function is used to invoke the `SetBalance` function on the `EtherBox` contract.
2. `SimpleCallWithGas`: This function creates an instance of the `EtherBox` contract and converts it into an address. Using this address, the `call` function is used to invoke the `SetBalance` function on `EtherBox`. Alongside the call, gas is also sent along, such that function execution can be completed if it needs more gas.
3. `SimpleCallWithGasAndValue`: This function creates an instance of the `EtherBox` contract and converts it into an address. Using this address, the `call` function is used to invoke the `SetBalance` function on `EtherBox`. Alongside the call, gas is also sent along, such that function execution can be completed if it needs more gas. Apart from gas, it is also possible to send Ether or wei to payable functions.

Take a look at the preceding functions in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
gma solidity ^0.4.17;

tract EtherBox {
 uint balance;

 function SetBalance() public {
     balance = balance + 10;
 }

 function GetBalance() public payable returns(uint) {
    return balance;
 }


tract UsingCall {
 function UsingCall() public payable  {
 }

 function SimpleCall() public returns (uint) {
     bool status = true;
     EtherBox eb = new EtherBox();
     address myaddr = address(eb);
     status =   myaddr.call(bytes4(sha3("SetBalance()")));
     return eb.GetBalance();
 }

 function SimpleCallwithGas() public returns (bool) {
     bool status = true;
     EtherBox eb = new EtherBox();
     address myaddr = address(eb);
     status =   myaddr.call.gas(200000)(bytes4(sha3("GetBalance()")));
     return status;
 }

 function SimpleCallwithGasAndValue() public returns (bool) {
     bool status = true;
     EtherBox eb = new EtherBox();
     address myaddr = address(eb);
     status =   myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance()")
     return status;
 }
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The callcode method

This function is deprecated and will not be discussed here. More information about `callcode` is available at http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The delegatecall method

This function is, again, a low-level function responsible for calling functions in another contract using the callers's state variables. Generally, it is used along with libraries in Solidity. More information about `delegatecall` is available at: http://solidity.readthedocs.io/en/develop/introduction-to-smart-contracts.html.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The fallback function

The fallback functions are a special type of function available only in Ethereum. Solidity helps in writing fallback functions. Imagine a situation where you, as a Solidity developer, are consuming a smart contract by invoking its functions. It is quite possible that you use a function name that does not exist within that contract. In such cases, the fallback function, as the name suggests, would automatically be invoked.

A fallback function is invoked when no function name matches the called function.

A fallback function does not have an identifier or function name. It is defined without a name. Since it cannot be called explicitly, it cannot accept any arguments or return any value. An example of a fallback function is as follows:

```
pragma solidity ^0.4.17;

contract FallbackFunction {

    function () {
        var a = 0;
    }

}
```

A fallback function can also be invoked when a contract receives any Ether. This usually happens using the `SendTransaction` function available in `web3` to send Ether from one account to a contract. However, in this case, the fallback function should be `payable`, otherwise it will not be able to accept the Ether and will raise an error.

The next important question to be answered is how much gas is needed to execute this function. Since it cannot be called explicitly, gas cannot be sent to this function. Instead, EVM provides a fixed stipend of 2,300 gas to this function. Any consumption of gas beyond this limit will raise an exception and the state will be rolled back after consuming all the gas that was sent along with the original function. It is therefore important to test your fallback function to ensure that it does not consume more than 2,300 gas.

It is also worth noting that fallback functions are one of the top causes of security lapses in smart contracts. It is very important to test this function from a security perspective before releasing a contract on production.

Let's now try to understand the fallback function with the help of some examples.

We will use the same example as we used for explaining the `call` function of the `address` data type.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

However, this time, we have implemented a `payable` fallback function in the `EtherBox` contract whose entire purpose is to raise an event and an additional function that calls an invalid function. The event is also declared within the function. We will look at events in more depth in the next chapter.

When you execute each of the methods in the `UsingCall` contract, you should notice that the fallback function is not invoked for any of the functions apart from one that does not call a correct function, as shown in the following screenshot:

```
pragma solidity ^0.4.17;

contract EtherBox {
    uint balance;
    event logme(string);

    function SetBalance() public {
        balance = balance + 10;
    }

    function GetBalance() public payable returns(uint) {
       return balance;
    }

    function() payable {
        logme("fallback called");
    }
}

contract UsingCall {
    function UsingCall() public payable  {
    }

    function SimpleCall() public returns (uint) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        status =   myaddr.call(bytes4(sha3("SetBalance()")));
        return eb.GetBalance();
    }

    function SimpleCallwithGas() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return status =   myaddr.call.gas(200000)(bytes4(sha3("GetBalance()")));
    }

    function SimpleCallwithGasAndValue() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return status =   myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance()")));
    }

    function SimpleCallwithGasAndValueWithWrongName() public returns (bool) {
        bool status = true;
        EtherBox eb = new EtherBox();
        address myaddr = address(eb);
        return myaddr.call.gas(200000).value(1)(bytes4(sha3("GetBalance1()")));
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Fallback functions are also invoked when using the `send` method, using the `web3` `SendTransaction` function, or the `transfer` method.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Summary

Once again, this was a heavy chapter that focused primarily on functions, including the `address` functions and the `pure`, `constant`, and `view` functions. The `address` functions can be intimidating, especially when you consider their multiple variations and their relationship with the fallback functions. If you are implementing a fallback function, remember to pay special attention to testing, especially from a security point of view. You should also pay special attention when using low-level Solidity functions such as `send`, `call`, and `transfer` as they invoke the fallback function implicitly. Always try using contract functions that use ABI as it ensures that the proper function, along with its data types, is being called.

In the next chapter, we will dive deep into the world of events, logging, and exception handling in Solidity. Stay tuned!

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Exceptions, Events, and Logging

Writing contracts is the fundamental purpose of Solidity. However, writing a contract demands sound error and exception handling. Errors and exceptions are the norm in programming and Solidity provides ample infrastructure for managing both. Writing robust contracts with proper error and exception management is one of the top best practices. Events are another important construct in Solidity. For all topics that we've discussed so far, we've seen a caller that invokes functions in contracts; however we have not discussed any mechanism through which a contract notifies its caller and others about changes in its state and otherwise. This is where events come in. Events are a part of event-driven programs where, based on changes within a program, it proactively notifies its caller about the changes. The caller is free to use this information or ignore it. Finally, both exceptions and events, to a large extent, use the logging feature provided by EVM.

In this chapter, we will cover the following topics:

- Understanding exception handling in Solidity
- Error handling with `require`
- Error handling with `assert`
- Error handling with `revert`
- Understanding events
- Declaring an event
- Using an event
- Writing to logs

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Error handling

Errors are often inadvertently introduced while writing contracts, so writing robust contracts is a good practice and should be followed. Errors are a fact of life in the programming world and writing error-free contracts is a desired skill. Errors can occur at design time or runtime. Solidity is compiled into bytecode and there are design-level checks for any syntax errors at design time while compiling. Runtime errors, however, are more difficult to catch and generally occur while executing contracts. It is important to test the contract for possible runtime errors, but it is more important to write defensive and robust contracts that take care of both design time and runtime errors.

Examples of runtime errors are out-of-gas errors, divide by zero errors, data type overflow errors, array-out-of-index errors, and so on.

Until version 4.10 of Solidity there was a single `throw` statement available for error handling. Developers had to write multiple `if...else` statements to check the values and throw in the case of an error. The `throw` statement consumes all the provided gas and reverts to the original state. This is not an ideal situation for architects and developers as unused gas should be returned back to the caller.

From version 4.10 of Solidity newer error handling constructs were introduced and `throw` was made obsolete. These were the `assert`, `require`, and `revert` statements. In this section, we will look into these error handling constructs.

It is worth noting that there are no `try..catch` statements or constructs to catch errors and exceptions.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The require statement

The word require denotes constraints. Declaring `require` statements means declaring prerequisites for running the function; in other words, it means declaring constraints that should be satisfied before executing the following lines of code.

The `require` statement takes in a single argument: a statement that evaluates to a `true` or `false` boolean value. If the evaluation of the statement is `false`, an exception is raised and execution is halted. The unused gas is returned to the caller and the state is reversed to the original. The `require` statement results in the `revert` opcode, which is responsible for reverting the state and returning unused gas.

The following code illustrates use of the `require` statement:

```
pragma solidity ^0.4.19;

contract RequireContract {

    function ValidInt8(uint _data) public returns(uint8){
        require(_data >= 0);
        require(_data <= 255);

        return uint8(_data);
    }

    function ShouldbeEven(uint _data) public returns(bool){
        require(_data % 2 == 0);
        return true;
    }

}
```

Let's take a look at the following functions depicted in the preceding screenshot:

1. `ValidInt8`: This function uses a couple of `require` statements. In constructs, a statement checks for values greater than or equal to zero. If this statement is `true`, execution passes to the next statement. If this statement is `false`, an exception is thrown and execution stops. The next `require` statement checks whether the value is less than or equal to `255`. If the argument is greater than `255`, the statement evaluates to `false` and throws an exception.
2. `ShouldbeEven`: This function is of a similar nature. In this function, `require` checks whether the incoming argument is even or odd. If the argument is even, execution passes to the next statement; otherwise an exception is thrown.

The `require` statement should be used for validating all arguments and values that are incoming to the function. This means that if another function from another contract or function in the same contract is called, the incoming value should also be checked using the `require` function. The `require` function should be used

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

to check the current state of variables before they are used. If `require` throws an exception, it should mean that the values passed to the function were not expected by the function and that the caller should modify the value before sending it to a contract.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The assert statement

The `assert` statement has a similar syntax to the `require` statement. If it accepts a statement, that should then evaluate to either a true or false value. Based on that, the execution will either move on to the next statement or throw an exception. The unused gas is not returned to the caller and instead the entire gas supply is consumed by `assert`. The state is reversed to original. The `assert` function results in invalid opcode, which is responsible for reverting the state and consuming all gas.

The function shown previously has been extended to include an addition to the existing variable. However, remember that adding two variables can result in an overflow exception. This is verified using the `assert` statement; if it returns `true`, the value is returned, otherwise the exception is thrown.

The following screenshot illustrates the use of the `assert` function:

```
pragma solidity ^0.4.19;

contract AssertContract {

    function ValidInt8(uint _data) public returns(uint8){
        require(_data >= 0);
        require(_data <= 255);

        uint8 value = 20;

        //checking datatype overflow
        assert (value + _data <= 255);

        return uint8(value + _data);
    }
}
```

While `require` should be used for values coming from the outside, `assert` should be used for validating the current state and condition of the function and contract before execution. Think of `assert` as working with runtime exceptions that you cannot predict. The `assert` statement should be used when you think that a current state has become inconsistent and that execution should not continue.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

The revert statement

The `revert` statement is very similar to the `require` function. However, it does not evaluate any statement and does not depend on any state or statements. Hitting a `revert` statement means an exception is thrown, along with the return of unused gas, and reverts to its original state.

In the following example, an exception is thrown when the incoming value is checked using the `if` condition; if the `if` condition evaluation results in `false`, it executes the `revert` function. This results in an exception and execution stops, as shown in the following screenshot:

```
pragma solidity ^0.4.19;

contract RevertContract {

    function ValidInt8(int _data) public returns(uint8){

        if(_data < 0 || _data > 255) {
            revert();
        }

        return uint8(_data);
    }
}
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Events and logging

We have seen the usage of events in previous chapters without going into any detail. In this section, however, we will look into events in more depth. Events are well known to event-driven programmers. Events refer to certain changes in contracts that raise events and notify each other such that they can act and execute other functions.

Events help us write asynchronous applications. Instead of continuously polling the Ethereum ledger for the existence of a transaction and then blocking with certain information, the same procedure can be implemented using events. This way, the Ethereum platform will inform the client if an event has been raised. This helps when writing modular code and also conserves resources.

Events are part of contract inheritance, where a child contract can invoke events. Event data is stored along with block data. The `logsBloom` value is the event data, as shown in the following screenshot:

Declaring events in Solidity is very similar to performing functions. However, events do not have any body. A simple event can be declared using the `event` keyword followed by an identifier and any parameters it wants to send along with the event as shown in the following code:

```
event LogFunctionFlow(string);
```

In the preceding line of code, `event` is the keyword used for declaring events followed by its name and a set of parameters that will be sent along with the event. Any string text can be sent with the `LogFunctionFlow` event.

Using an event is quite simple. Simply invoke an event using its name and pass on the arguments it expects. For the `LogFunctionFlow` event, the invocation would look as follows which is similar to a function call with parameters:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
LogFunctionFlow("I am within function x");
```

The following code snippet shows an event in use. In this example, an event, `LogFunctionFlow` , is declared with a string as its sole parameter. The same event is invoked multiple times from the `ValidInt8` function, providing text information during various stages within the function:

```
pragma solidity ^0.4.19;

contract EventContract {

    event LogFunctionFlow(string);

    function ValidInt8(int _data) public returns(uint8){
        LogFunctionFlow("Within function ValidInt8");

        if(_data < 0 || _data > 255) {
            revert();
        }

        LogFunctionFlow("Value is within expected range");
        LogFunctionFlow("Returning value from function");

        return uint8(_data);
    }
}
```

Executing this contract in Remix shows the result, which contains three logs with event information as shown in the following screenshot:

```
[
        {
                "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918
                "event": "LogFunctionFlow",
                "args": [
                        "Within function ValidInt8"
                ]
        },
        {
                "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918
                "event": "LogFunctionFlow",
                "args": [
                        "Value is within expected range"
                ]
        },
        {
                "topic": "b5b850034705238ab6360bdc803e9e3dcaaf926c812b20193e2e99a5918
                "event": "LogFunctionFlow",
                "args": [
                        "Returning value from function"
                ]
        }
]
```

Events can also be watched from custom applications and decentralized applications using `web3`.

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

Events can be filtered using parameters names.

The following two methods allow us to watch for events:

1. **Watching individual events**: In this method, using `web3`, individual events from contracts can be watched and tracked. When the exact event is fired from a contract, it helps execute a function in the `web3` client. An example of watching an individual event is shown in the following screenshot. Here, `ageRead` is the name of the event we are interested in and watching for. We read `fromBlock` number `25000` until the `latest` block. First, a reference to the `ageRead` event is made and a watcher is added to the reference. The watcher takes a `promise` function that is executed whenever the `ageRead` event is fired:

```
var myEvent = instance.ageRead({fromBlock: 25000, toBlock: 'latest'});
myEvent.watch(function(error, result){
    if(error) {
        console.log(error);
    }
    console.log(result.args)
});
```

1. **Watching all events**: In this method, using `web3` all events from contracts can be watched and tracked. When any event is fired from a contract, it notifies and helps to execute a function in the `web3` client in response. In this case, the event can be filtered using an event name. An example of watching all events is shown in the following screenshot. Here, we are interested in and watching for any event from a contract. We read `fromBlock` number `25000` until the `latest` block. First, a reference to `allEvents` is made and a watcher is added to the reference. The watcher then takes a `promise` function that is executed whenever any event is fired:

```
var myEvent = instance.allEvents({fromBlock: 24000, toBlock: 'latest'});
myEvent.watch(function(error, result){
    if(error) {
        console.log(error);
    }
    console.log(result)
});
```

The value in the `result` object from the event is shown in the following screenshot:

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863

```
ddress: '0x600c320dd768fb55f03748d4d4028db2cafc06a9',
lockNumber: 24864,
ransactionHash: '0x38ca3d4b40f8e75d27ab3950234d837e96dbdd086178f139c4e675dc6531ee15',
ransactionIndex: 0,
lockHash: '0x778b9a9a89b4609475dcc52d4c60de44254c24f8356b4886496488f57761ca0c',
ogIndex: 0,
emoved: false,
vent: 'ageRead',
rgs: { '': '33' } }
```

Copyright © 2018. Packt Publishing. All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 2/28/2023 7:09 AM via THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY
AN: 1801021 ; Ritesh Modi.; Solidity Programming Essentials : A Beginner's Guide to Build Smart Contracts for Ethereum and Blockchain
Account: ns249863