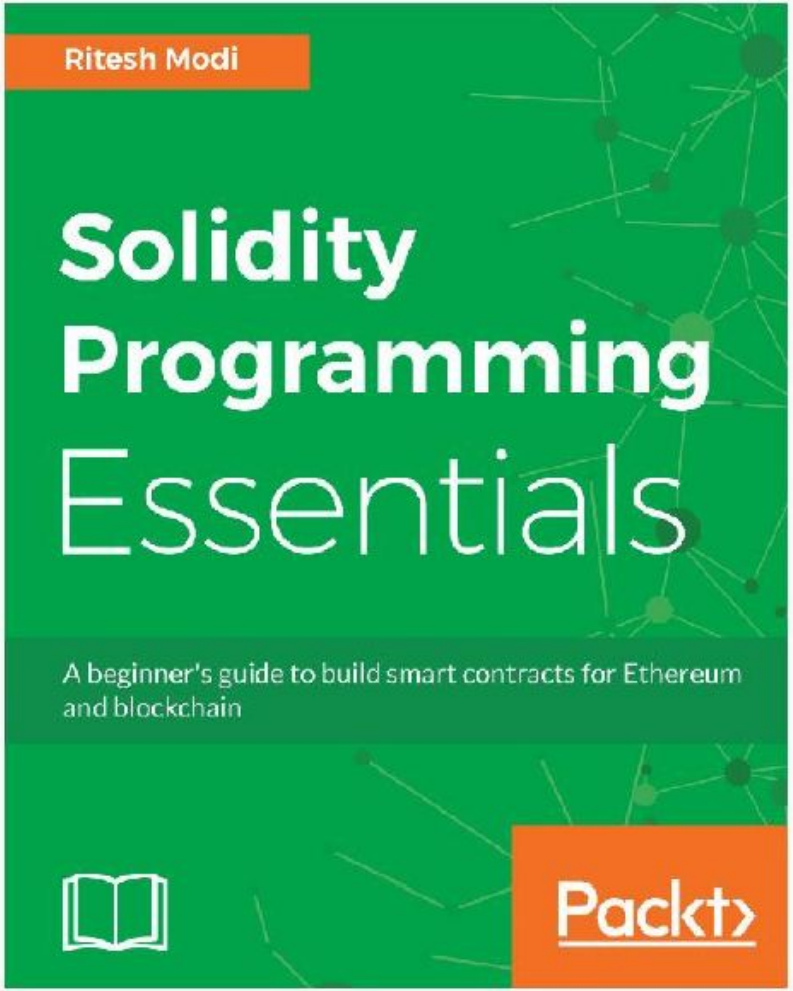
SOLIDITY
Part 2

Zeeshan Hanif

Book we will follow



Prerequisite

1. Understanding of Blockchain
2. Understanding of Ethereum Blockchain
3. Fundamentals of Programming

Object Oriented Programming

Object Oriented Programming

1. As we are using software development to automate our manual tasks
2. To convert our real world scenario/situation into software application
3. Therefore, almost all features of programming languages are taken from real world
4. Object Oriented programming features are also taken from real world, the way we have objects around us in real world.

Object Oriented Programming

Object Oriented Programming (OOP) is a software design pattern that allows you to think about problems in terms of objects and their interactions.

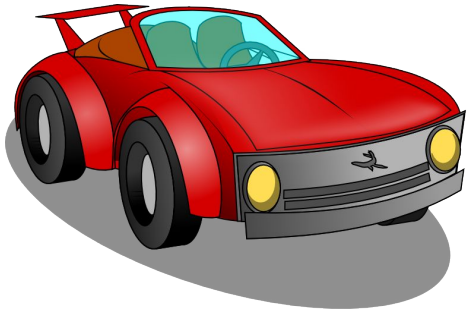
Object Oriented Programming

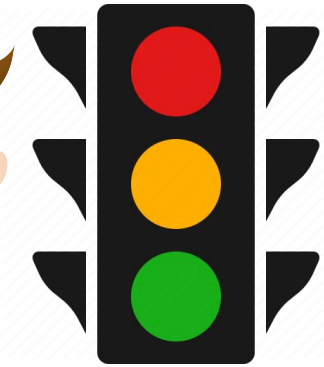
Object-oriented programming (OOP) is a programming language model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

What is an Object?

It is a basic unit of Object Oriented Programming and represents the real life entities.

Everything is an Object





What is an Object?

1. If we consider the real-world, we can find many objects around us, cars, dogs, humans, bicycle, etc.
2. These real-world objects share two characteristics: they all have state and they all have behavior.
3. If we consider a dog, then its state are name, breed, color, and the behaviors are barking, wagging the tail, running.
4. Bicycle have state current gear, two wheels, number of gears, and the behavior are braking, accelerating, slowing down, changing gears.

What is an Software Object?

1. Software objects are modeled after real-world objects in that they, too, have state and behavior.
2. A software object maintains its state in variables and implements its behavior with methods.
3. So in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

What is an Software Object?

1. In Object Oriented terms, an Object is a software unit of variables (properties) and methods (functions).
2. These objects are often used to model the real-world objects that you find in everyday life.
3. An Object's method provide the only way to access the data.
4. From a programming point of view, an object can be a data structure, a variable or a function. It has a memory location allocated.

What is Class?

A class is an entity that determines how an object will behave and what the object will contain. In other words, it is a blueprint or a set of instruction to build a specific type of object.

What is Class?

1. A class is a blueprint or template or set of instructions to build a specific type of object.
2. Every object is built from a class.
3. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.

Difference Between Object & Class

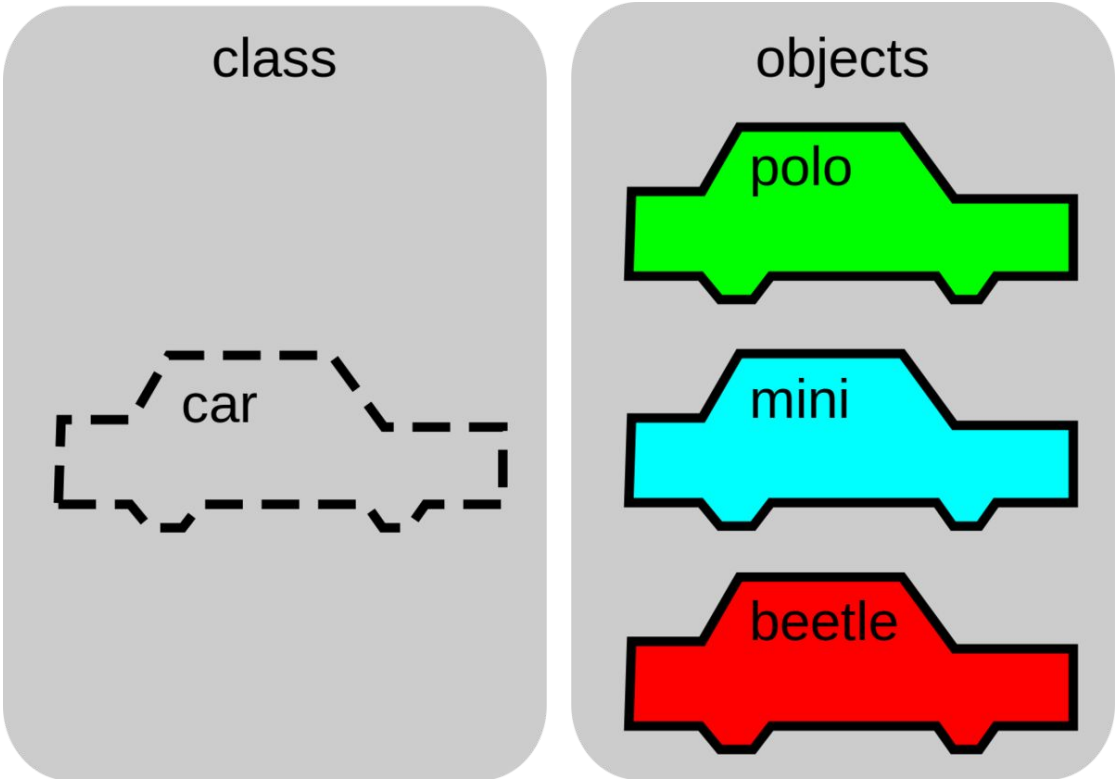
Difference Between Object & Class

1. A class is a blueprint or template that defines the variables and the methods (functions) common to all objects of a certain kind.
2. An object is a specimen of a class.
3. An Object within Object Oriented Programming is an instance of a Class.
4. Each Object built from the same set of blueprints (Class) and therefore contains the same components .

Example

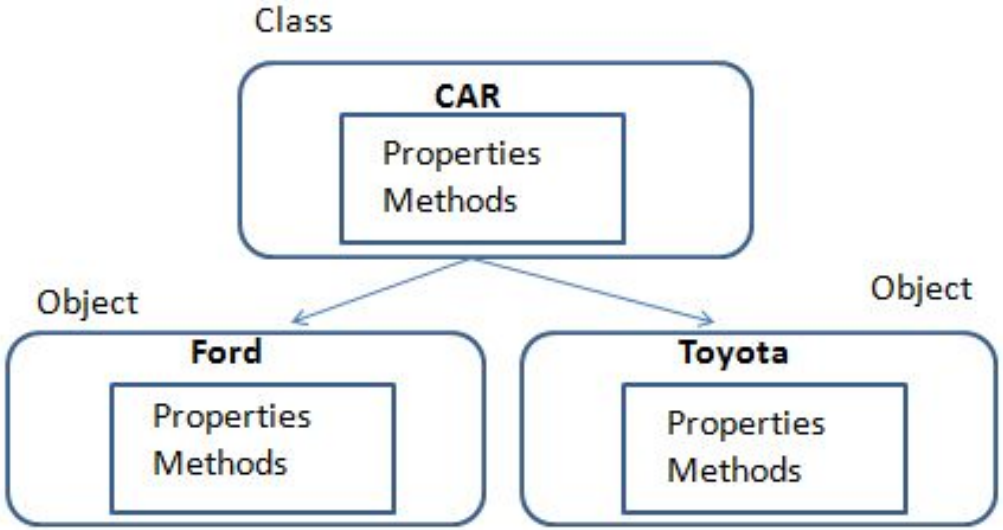
1. Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc.
2. So here, Car is the class and wheels, speed limits, mileage are their properties.

Difference Between Object & Class



Example

1. Ford and Toyota share certain similar features and hence can be grouped to form the Object of the Class Car.

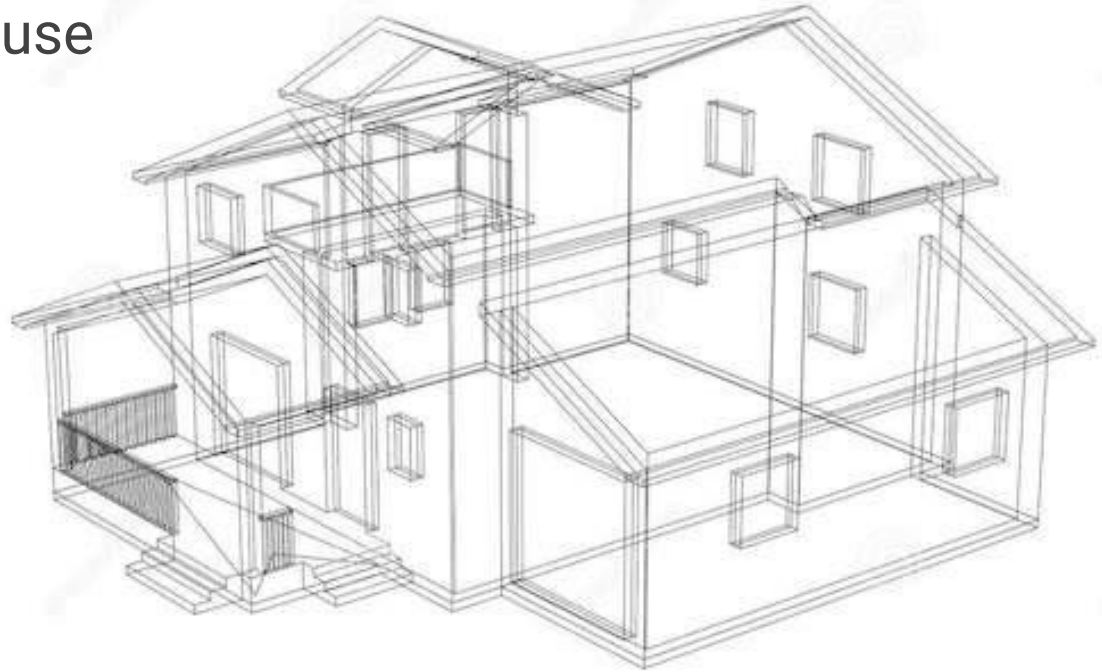


Example of House

1. Think about classes, instances, and instantiation like creating a house.
2. A **Class** is like a design or blueprint of House.
3. It’s not a real physical house but just a specification with all the properties a house will have — rooms, floors, area etc.

Example of House

Class of house



Example of House

1. An **object** is a particular instance of the class. Two different objects of a class can have different values for properties and are (mostly)independent of each other.
2. Taking the example of House class, if there are 2 objects of class House — they can have a different number of rooms, area, address, # of floors etc.
3. Think of an object as an actual physical house

Example of House

Instance of house

Principles of Object-Oriented Programming

1. Encapsulation
2. Abstraction
3. Inheritance
4. Polymorphism

Encapsulation

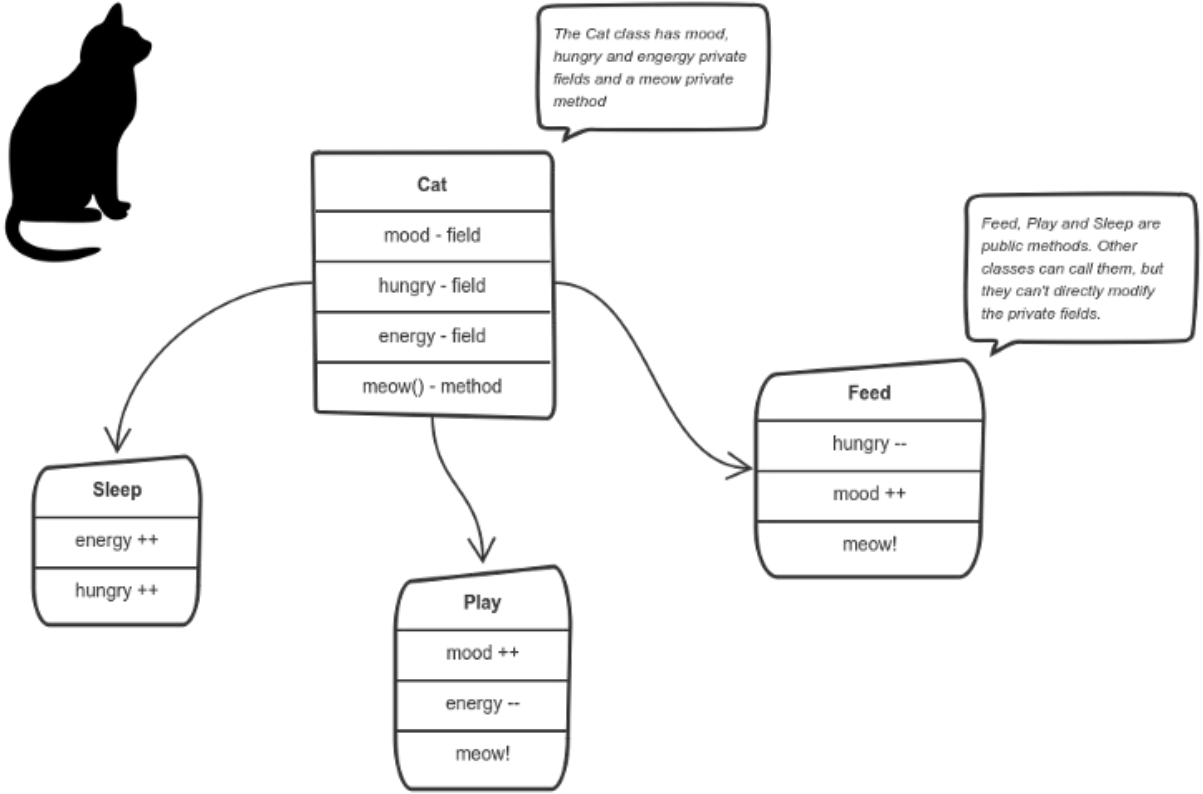
Encapsulation

1. Encapsulation is an object-oriented programming concept that binds together the data and functions that manipulate the data as a single unit
2. It also means to hide your data in order to make it safe from any modification.
3. Another way to think about encapsulation is, it is a protective shield that prevents the data from being accessed by the code outside this shield.

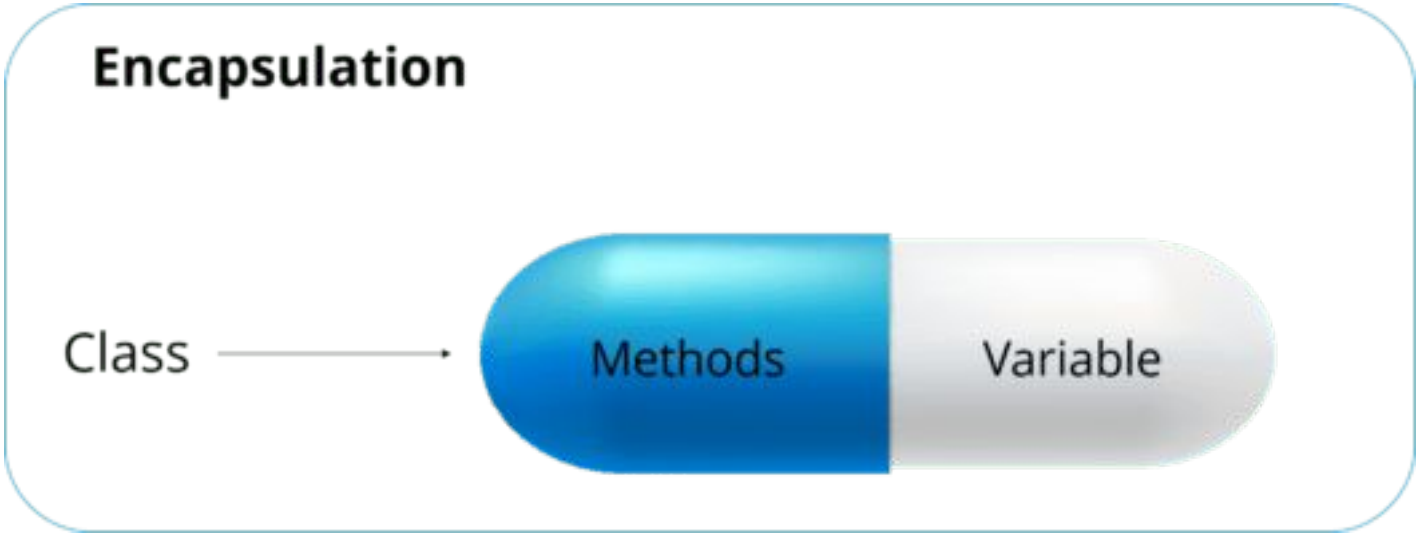
Encapsulation

4. Encapsulation is achieved when each object keeps its state private, inside a class.
5. The object manages its own state via methods — and no other class can touch it unless explicitly allowed. If you want to communicate with the object, you should use the methods provided. But (by default), you can’t change the state.

Encapsulation



Encapsulation



Abstraction

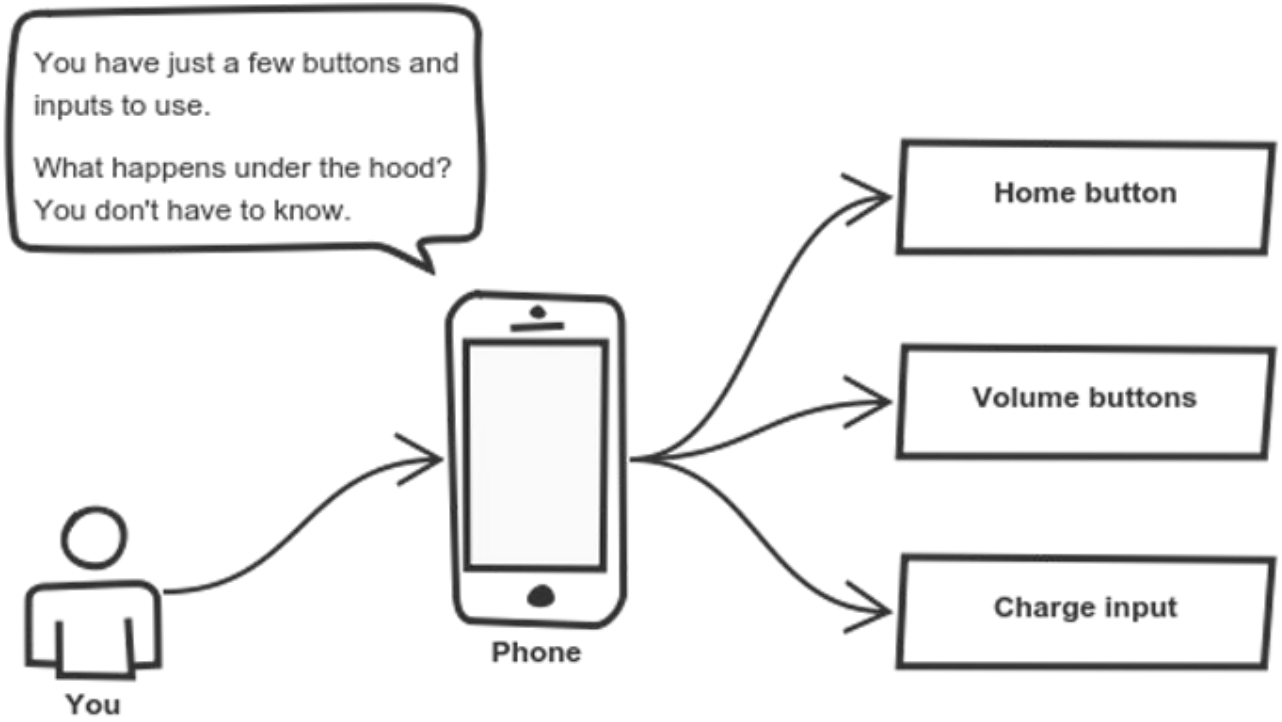
Abstraction

1. Abstraction can be thought of as a natural extension of encapsulation.
2. Applying abstraction means that each object should only expose a high-level mechanism for using it.
3. This mechanism should hide internal implementation details. And it should only reveal operations relevant for the other objects.

Abstraction

4. Data Abstraction may also be defined as the process of identifying only the required characteristics of an object ignoring the irrelevant details.
5. Think of a coffee machine. It does a lot of stuff and makes quirky noises under the hood. But all you have to do is put in coffee and press a button.
6. Think of it as a small set of public methods which any other class can call without “knowing” how they work.

Abstraction



Inheritance

Inheritance

1. Objects are often very similar. They share common logic. But they’re not entirely the same.
2. So how do we reuse the common logic and extract the unique logic into a separate class? One way to achieve this is **inheritance**.
3. Inheritance is one such concept where the properties of one class can be inherited by the other. It helps to reuse the code and establish a relationship between different classes.

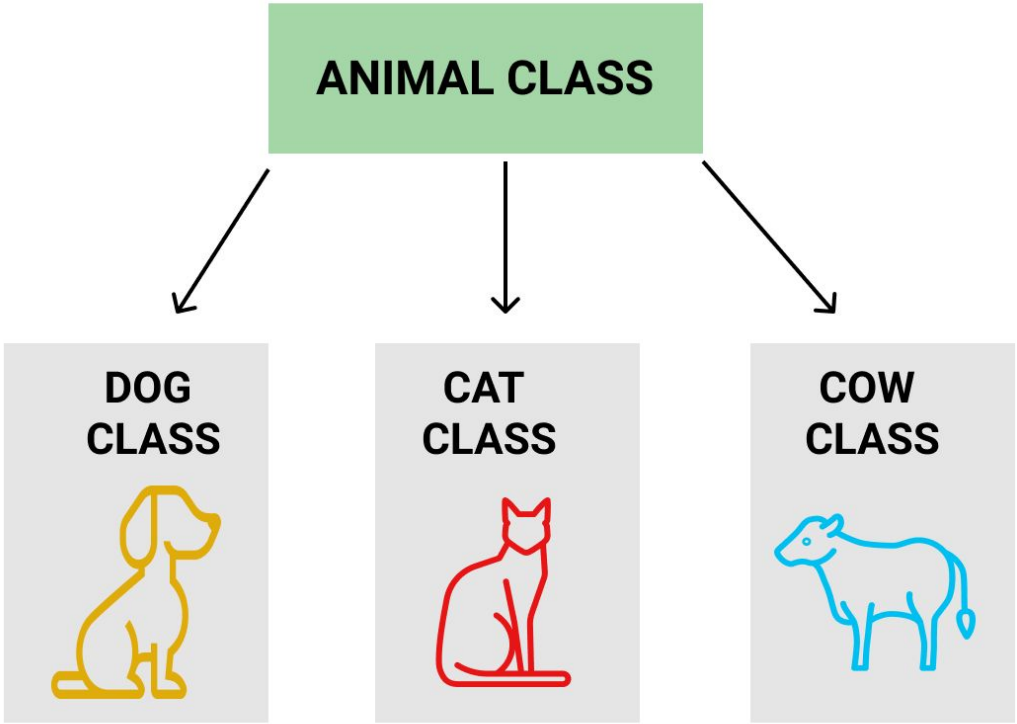
Inheritance

4. It means that you create a (child) class by deriving from another (parent) class. This way, we form a hierarchy.
5. The child class reuses all fields and methods of the parent class (common part) and can implement its own (unique part).
6. This way, each class adds only what is necessary for it while reusing common logic with the parent classes.

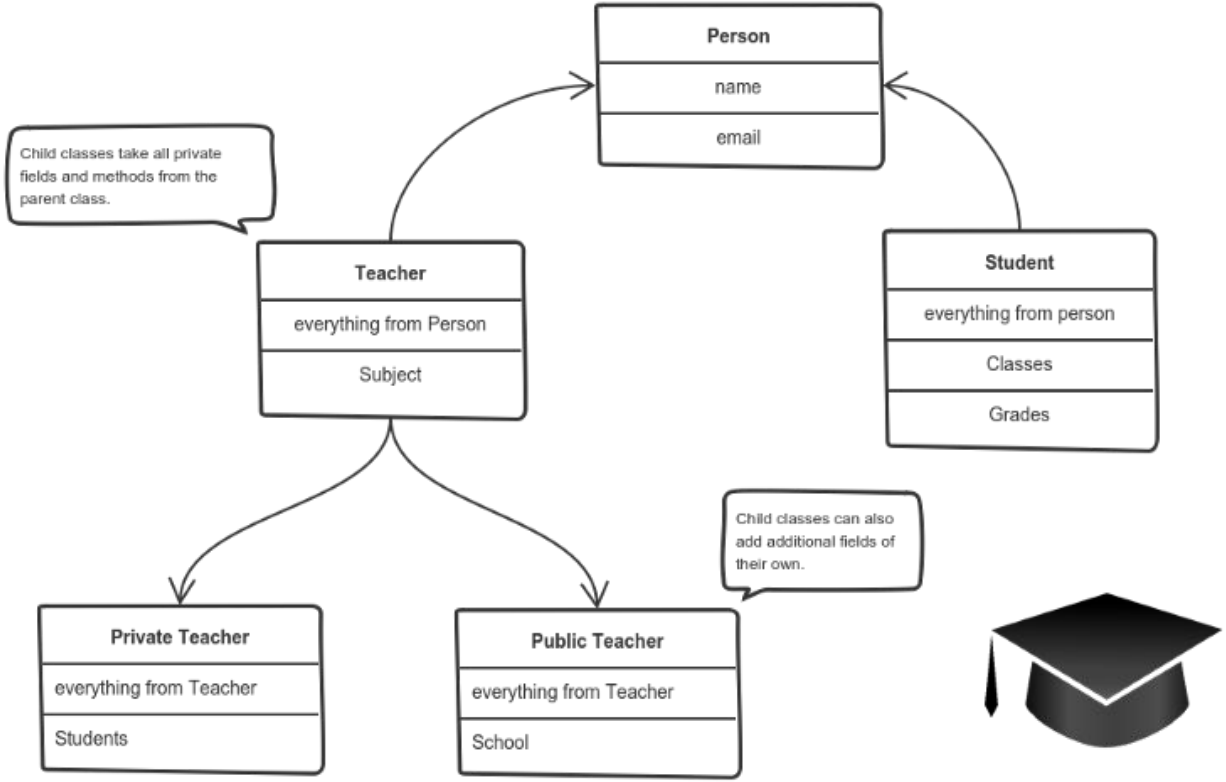
Inheritance

7. Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class.
8. By doing this, we are reusing the fields and methods of the existing class.

Inheritance



Inheritance

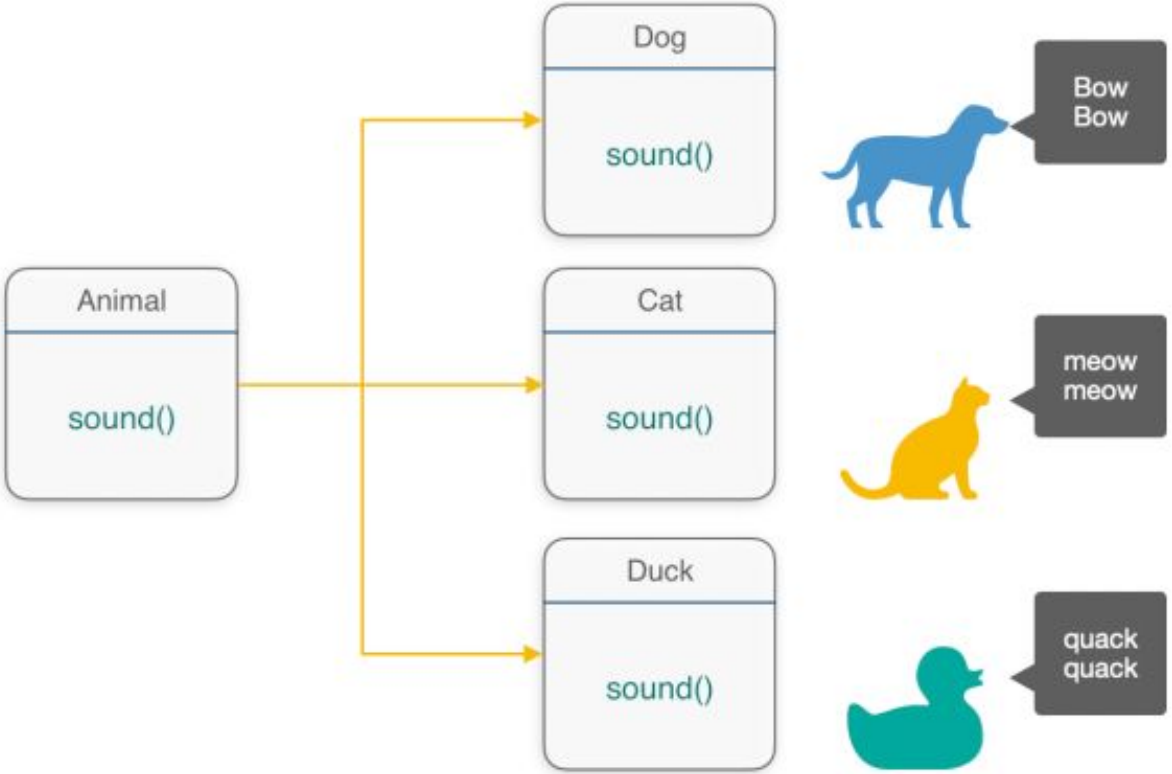


Polymorphism

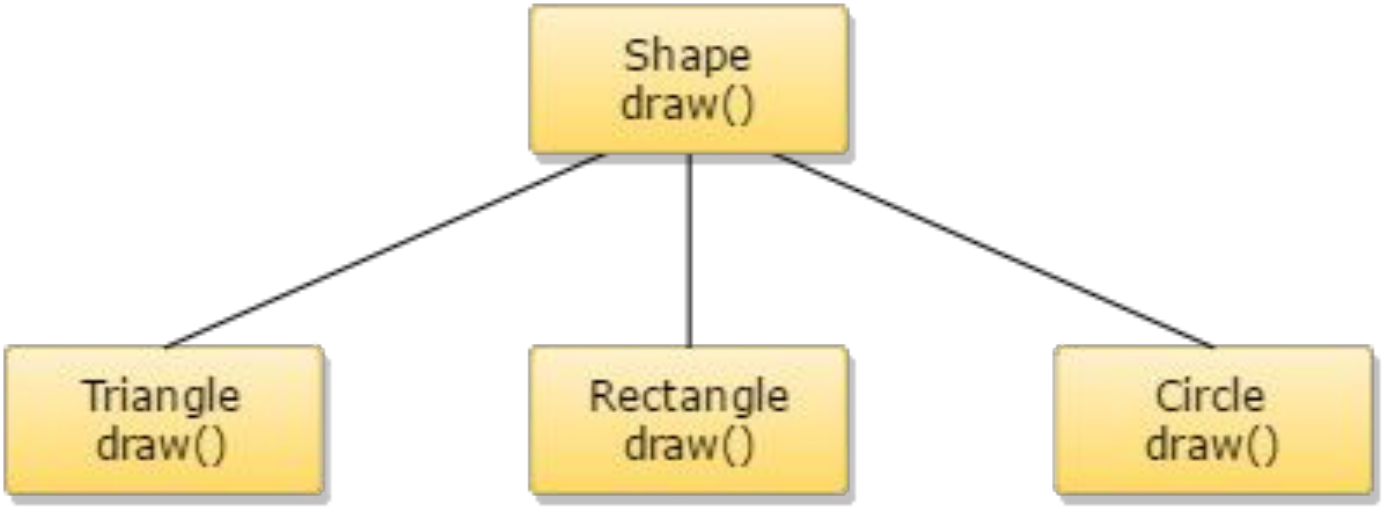
Polymorphism

1. Polymorphism means “many shapes” in Greek.
2. It is the ability of a variable, function or object to take on multiple forms. In other words, polymorphism allows you define one interface or method and have multiple implementations.
3. In object-oriented programming, polymorphism refers to a programming language's ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.

Polymorphism



Polymorphism



Polymorphism

Polymorphism

1. Polymorphism gives a way to use a class exactly like its parent so there's no confusion with mixing types. But each child class keeps its own methods as they are.
2. This typically happens by defining a (parent) interface to be reused. It outlines a bunch of common methods. Then, each child class implements its own version of these methods.

Polymorphism

1. For example, given a base class Animal, polymorphism enables the programmer to define different sound methods for any number of derived classes, such as Dog, Cat and Duck. No matter what type of animal object is, applying the sound method to it will return the correct results. Polymorphism is considered to be a requirement of any true object-oriented programming language (OOPL).

Chapter 6
Writing Smart Contracts

Based on Solidity 0.6.0

Smart contracts

1. The most important feature of the Ethereum blockchain is its ability to execute software code known as smart contracts.
2. With the Ethereum blockchain, you do not need to trust anyone to correctly execute your code.
3. Instead, a community of network participants (Ethereum miners) will all execute your smart contract code and reach a consensus that the results are correct.

Smart contracts

4. Smart contracts are, essentially, code segments or programs that are deployed and executed in EVM.
5. A contract is a term generally used in the legal world and has little relevance in the programming world.
6. Writing a smart contract in Solidity does not mean writing a legal contract.
7. The code controls the execution, and transactions are trackable and irreversible.

Agreement in form of computer code.

A smart contract is an agreement between two people in the form of computer code. They run on the blockchain, so they are stored on a public database and cannot be changed.

No Third Party Required

The transactions that happen in a smart contract processed by the blockchain, which means they can be sent automatically without a third party. This means there is no one to rely on!

Trustless

The transactions only happen when the conditions in the agreement are met — there is no third party, so there are no issues with trust.

Smart contracts

1. A smart contract is very similar to a C++, Java, or C# class.
2. Just as a class is composed of state (variables) and behaviors (methods), contracts contain state variables and functions.
3. 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.

Writing a simple contract

1. A contract is declared using the contract keyword along with an identifier
2. Within the brackets comes the declaration of state variables and function definitions which we are already covered in detail

```
contract Student {

}
```

Creating Contracts

Two ways of creating and using a contract in Solidity

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

1) Creating Contract using the new keyword

1. The new keyword in Solidity deploys and creates a new contract instance
2. It initializes the contract instance by:
    a. Deploying the contract
    b. Initializing the state variables
    c. Running its constructor
    d. Setting the nonce value to one
    e. Returns the address of the instance to the caller.

1) Creating Contract using the new keyword

3. Deploying a contract involves checking whether
    a. The requestor has provided enough gas to complete deployment
    b. Generating a new account/address for contract deployment using the requestor's address and nonce value
    c. Passing on any Ether sent along with it.
4. Everytime we use new keyword to create object it deploys new contract which has its own ethereum address

1) Creating Contract using the new keyword

1. In next example we will create two contract
   a. Student
   b. Client
2. Client contract will create new instance of Student contract using new keyword

1) Creating Contract using the new keyword

```
contract Student {
    uint private age;
    function getAge() public returns (uint) {
        return age;
    }
    function setAge(uint a) public {
        age = a;
    }
}
```

1) Creating Contract using the new keyword

```
contract Client {
    function createObjects() public {
        Student st = new Student();
        st.setAge(45);
    }
}
```

2) Creating Contract using address of a contract

1. This method of creating a contract uses the address of an existing, deployed contract.
2. No new instance is created; rather, an existing instance is reused.
3. A reference to the existing contract is made using its address.

2) Creating Contract using address of a contract

1. In next example we will create two contract
   a. Student
   b. Client
2. Client contract will use already known address of Student contract to reference it.
3. It will not create new instance
4. It uses address data type and casting the actual address to the Student contract type.

2) Creating Contract using address of a contract

```
contract Student {
    uint private age;
    function getAge() public returns (uint) {
        return age;
    }
    function setAge(uint a) public {
        age = a;
    }
}
```

2) Creating Contract using address of a contract

```
contract Client {
    address obj = 0x692a70D2e424a56D2C6C27aA97D1a86395877b3A;
     function createObjects() public {
          Student st = Student(obj);
          st.setAge(45);
     }
}
```

Constructors

1. The constructor is executed once while deploying the contract.
2. Constructors are optional in Solidity and the compiler induces a default constructor when no constructor is explicitly defined.
3. In solidity creating new instance means deploying new contract therefore constructor is called once when contract deployed
4. Constructors should be used for initializing state variables and, generally, writing extensive Solidity code should be avoided.

Constructors

5. There can be only one constructor in a contract
6. Constructors can take parameters and arguments should be supplied while deploying the contract
7. Constructor should be created with keyword constructor and it does not return any data explicitly.
8. A constructor can be either public or internal, from a visibility point of view.

Constructors

```
contract Student {
    uint private age;
    constructor(uint _age) public {
        age = _age;
    }
    function getAge() public returns (uint) {
        return age;
    }
}
```

Constructors

```
contract Client {
    function createObjects() public {
        Student st  = new Student(5);
        st.getAge();
    }
}
```

Constructors

```
contract Client {
    function createObjects() public {
          // Compile time error,
          // zero argument constructor not available
        Student st  = new Student();
        st.getAge();
    }
}
```

Inheritance

Inheritance

1. Inheritance is the process of defining multiple contracts that are related to each other through parent-child relationships.
2. The contract that is inherited is called the **parent contract** and the contract that inherits is called the **child contract**.
3. Similarly, the contract has a parent known as the **derived class** and the parent contract is known as a **base contract**.
4. Inheritance is mostly about code-reusability.

Inheritance

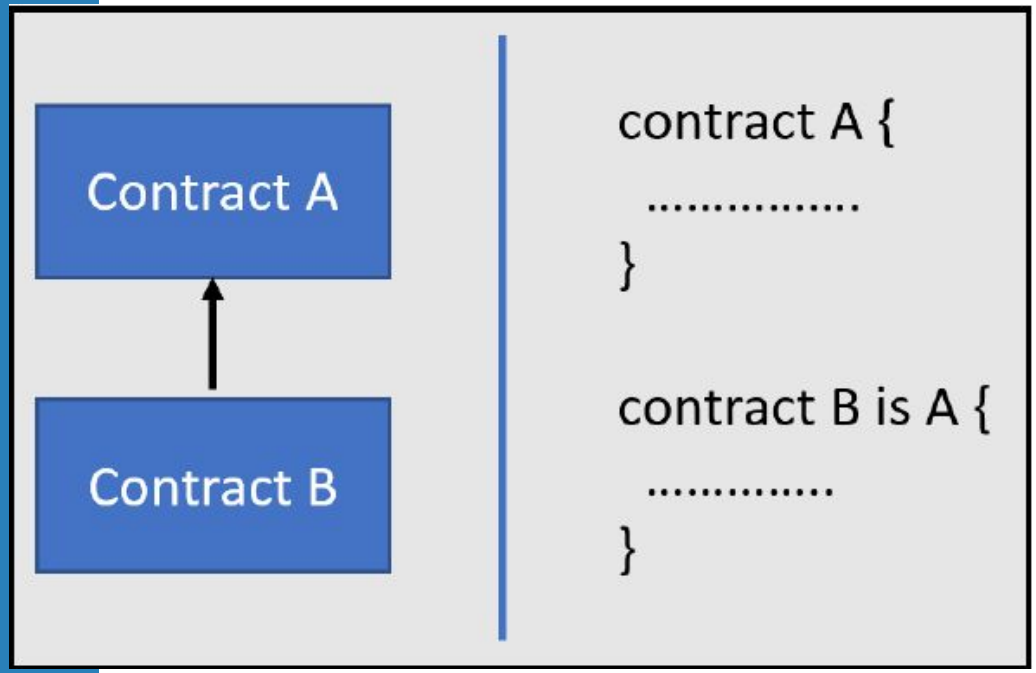
5. There is a is-a relationship between base and derived contracts and all public and internal scoped functions and state variables are available to derived contracts.
6. Solidity compiler copies the base contract bytecode into derived contract bytecode.
7. The ***is*** keyword is used to inherit the base contract in the derived contract.

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.”

Single inheritance

Single inheritance helps in inheriting the variables, functions, modifiers, and events of base contracts into the derived class.



Inheritance -- Example 1a

```
contract Human {
    uint internal age;
    function setAge(uint a) public {
        age = a;
    }}
contract Student is Human {
    function getAge() public returns (uint) {
        return age;
    }}
```

Inheritance -- Example 1b

1. setAge function is inherited from parent Human contract

```
contract Client {
    function createObjects() public {
        Student st  = new Student();
        st.setAge(20);
        st.getAge();
    }
}
```

Inheritance -- State Variable shadowing is not allowed

1. State variable shadowing is not allowed.
2. A derived contract can only declare a state variable x, if there is no visible state variable with the same name in any of its bases.
3. If parent contract has state variable with **public** or **internal** visibility, then child contact cannot declare variable with same name
4. If parent contract has state variable with private visibility, then child contact can declare variable with same name

Inheritance -- Example 2

```
contract Human {
    uint internal age;
    function setAge(uint a) public {
        age = a;
}}
contract Student is Human {
    uint internal age;  // Compile time Error, Not allowed
    function getAge() public returns (uint) {
        return age;
}}
```

Inheritance -- Example 3

```
contract Human {
    uint private age;// Parent class have private state
    function setAge(uint a) public {
        age = a;
}}
contract Student is Human {
    uint internal age;//Works fine- public, internal, private
    function getAge() public returns (uint) {
        return age;
}}
```

Inheritance -- No constructor in child and parent

1. If there is no constructor in parent and child contract then child contract’s object can be created with default zero argument constructor.

Inheritance -- Example 4a

```
contract Human {
    uint public age;
}
contract Student is Human {
    function getAge() public returns (uint) {
        return age;
}}
```

Inheritance -- Example 4b

1. Create object with default constructor works fine

```
Student st  = new Student();
st.getAge();
```

Inheritance -- Default constructor in parent

1. If there is constructor in parent contract with default zero argument then child contract’s object can be created with default zero argument constructor

Inheritance -- Example 5a

```
contract Human {
    uint public age;
    constructor () public {   }
}
contract Student is Human {
    function getAge() public returns (uint) {
        return age;
}}
```

Inheritance -- Example 5b

1. Create object with default constructor works fine

```
Student st  = new Student();
st.getAge();
```

Inheritance -- Constructor with parameter in parent

1. If there is constructor in parent contract with one or more parameter then child must call parent constructor and provide required arguments
2. Child contract will not compile if it will not provide call to parent constructor

Inheritance -- Example 6a

```
contract Human {
    uint public age;
    constructor (uint _a) public {
        age = _a;
    }}
contract Student is Human {
    constructor(uint _b) Human(_b) public {}
    function getAge() public returns (uint) {
        return age;
    }}
```

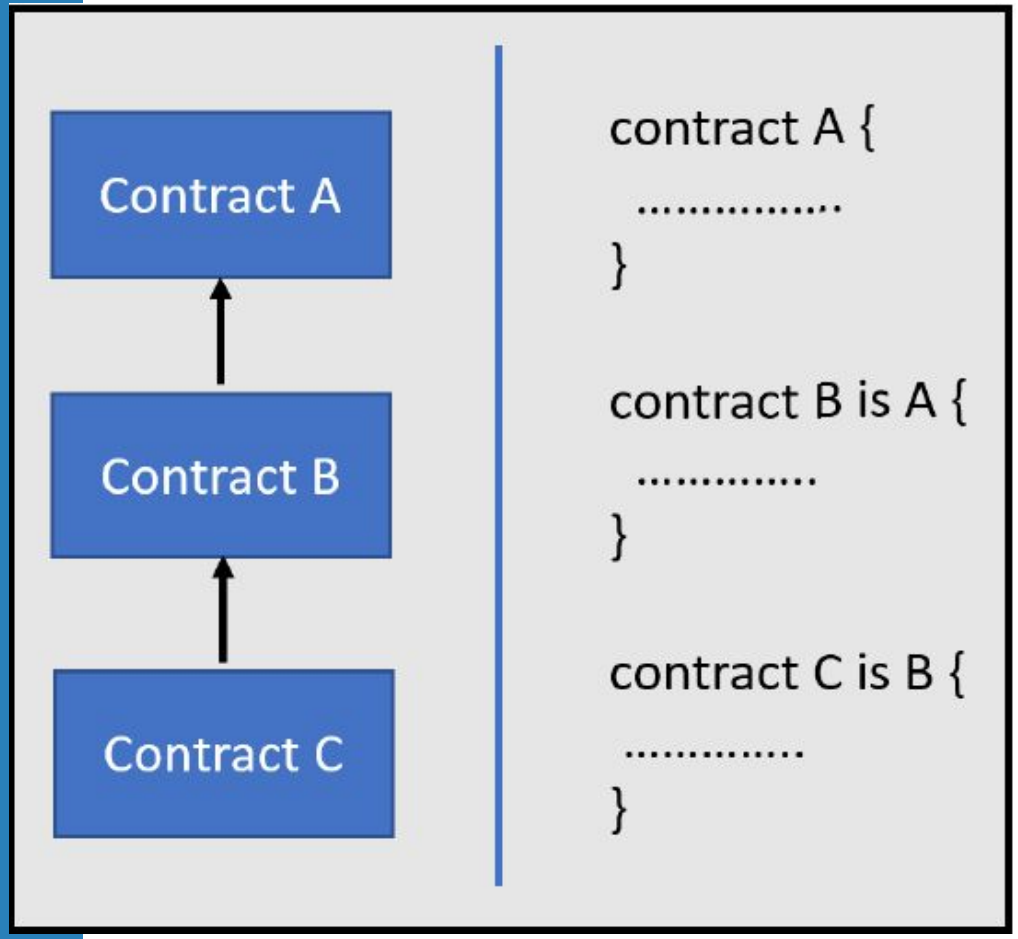
Inheritance -- Example 6b

1. Create object with parameter will call constructor of parent contract and set value of age in parent contract

```
Student st  = new Student(5);
st.getAge();// returns 5
```

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.



Multi-level Inheritance -- Example 7a

```
contract Human {
    uint internal age;
    string internal name;
    function setAge(uint _a) public {
        age = _a;
    }
    function getName() public returns (string memory){
        return name;
    }
}
```

Multi-level Inheritance -- Example 7b

```
contract Student is Human {
    function getAge() public returns (uint) {
        return age;
    }
    function setName(string memory _name) public {
        name = _name;
    }
}
```

Multi-level Inheritance -- Example 7c

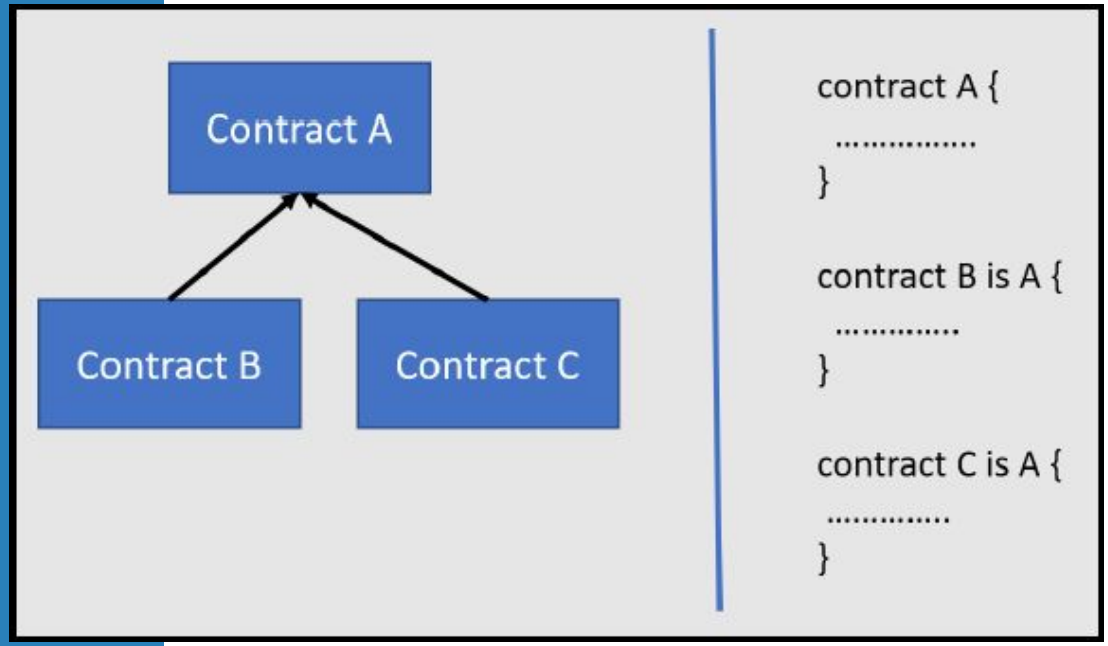
```
contract BlockchainStudent is Student {
    function getCourseName() public returns (string memory) {
        return "Blockchain";
    }
}
```

Multi-level Inheritance -- Example 7d

```
contract Client {
    function createObjects() public {
        Human h = new Human();
        Student s  = new Student();
        BlockchainStudent bs = new BlockchainStudent();
        bs.getName();
        bs.getAge();
        bs.getCourseName();
    }
}
```

Hierarchical Inheritance

Hierarchical inheritance is again similar to simple inheritance. Here, however, a single contract acts as a base contract for multiple derived contracts.



Hierarchical Inheritance -- Example 8a

```
contract Human {
    uint internal age;
    function setAge(uint _a) public {
        age = _a;
    }
    function getAge() public returns (uint) {
        return age;
    }
}
```

Hierarchical Inheritance -- Example 8b

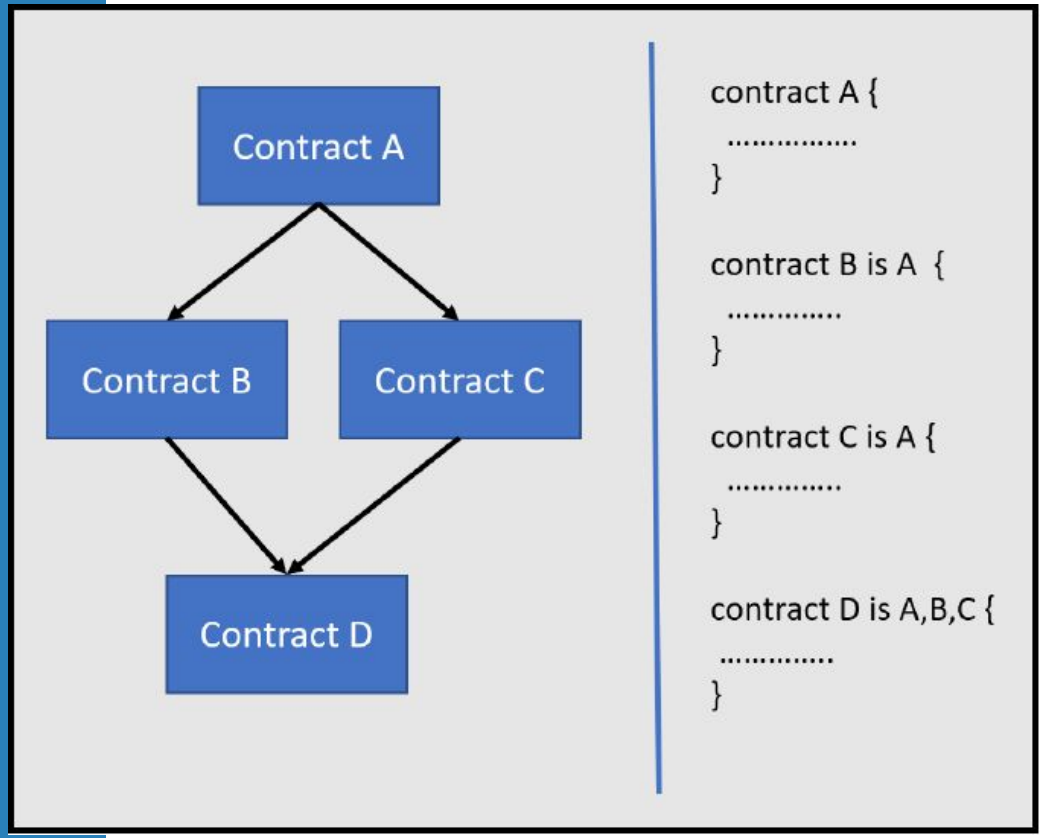
```
contract Student is Human {
    function getCourseName() public returns (string memory){
        return "Blockchain";
    }
}
contract Teacher is Human {
    function getQualification() public returns (string memory){
        return "Masters";
    }
}
```

Hierarchical Inheritance -- Example 8c

```
contract Client {
    function createObjects() public {
        Human h = new Human();
        Student s  = new Student();
        Teacher t = new Teacher();
        s.getCourseName();
        t.getQualification();
    }
}
```

Multiple Inheritance

Solidity supports multiple inheritance.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



Multiple Inheritance -- Example 9a

```
contract Human {
}
contract Student is Human {
}
contract Teacher is Human {
}
contract BlockchainStudentAndTeacher is Student, Teacher {
}
```

Solidity Part 2 Ends