Dark blue - didn't understand that point
Green - explained in my language
Red - notes

Secureum 101

1. Solidity is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language.

2. The best-practices for layout within a contract is the following order: state variables, events, modifiers, constructor and functions.

3. SPDX License Identifier: Solidity source files are recommended to start with a comment indicating its license e.g.:"// SPDX-License-Identifier: MIT", where the compiler includes the supplied string in the bytecode metadata to make it machine readable. SPDX stands for Software Package Data Exchange

4. Pragmas: The pragma keyword is used to enable certain compiler features or checks. A pragma directive is always local to a source file, so you have to add the pragma to all your files if you want to enable it in your whole project. If you import another file, the pragma from that file does not automatically apply to the importing file. There are two types: 1) Version: a) Compiler version b) ABI Coder version 2) Experimental: a) SMTChecker

5. Version Pragma: This indicates the specific Solidity compiler version to be used for that source file and is used as follows: "pragma solidity x.y.z;" where x.y.z indicates the version of the compiler.
   a. Using the version pragma does not change the version of the compiler. It also does not enable or disable features of the compiler. It just instructs the compiler to check whether its version matches the one required by the pragma. If it does not match, the compiler issues an error.
   b. The latest compiler versions are in the 0.8.z range
   c. A different y in x.y.z indicates breaking changes e.g. 0.6.0 introduces breaking changes over 0.5.z. A different z in x.y.z indicates bug fixes.
   d. A '^' symbol prefixed to x.y.z in the pragma indicates that the source file may be compiled only from versions starting with x.y.z until x.(y+1).z. For e.g., "pragma solidity ^0.8.3;" indicates that source file may be compiled with compiler version starting from 0.8.3 until any 0.8.z but not 0.9.z. This is known as a "floating pragma."
   e. Complex pragmas are also possible using '>','>=','<' and '<=' symbols to combine multiple versions e.g. "pragma solidity >=0.8.0 <0.8.3;"

6. ABI Coder Pragma: This indicates the choice between the two implementations of the ABI encoder and decoder: "pragma abicoder v1;" or "pragma abicoder v2;"
   a. The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. It might produce less optimal code and has not received as much testing as the old encoder. This is activated by default.

b. The set of types supported by the new encoder is a strict superset of the ones supported by the old one. (iska matlab abicoder v2 = abicoder v1 + new feature)
c. Contracts that use it can interact with ones that do not without limitations. The reverse is possible only as long as the non-abicoder v2 contract does not try to make calls that would require decoding types only supported by the new encoder. The compiler can detect this and will issue an error. Simply enabling abicoder v2 for your contract is enough to make the error go away. (sidhi si baat hai bhai v2 wala v1 se interact kar sakta hai. v1 bhi v2 interact kar sakta par v1 wala contract ko Dhyan rakhna hoga ki wo koi aisa call na karde jisko decode sirf v2 hi kar paye.)
d. This pragma applies to all the code defined in the file where it is activated, regardless of where that code ends up eventually. This means that a contract whose source file is selected to compile with ABI coder v1 can still contain code that uses the new encoder by inheriting it from another contract. This is allowed if the new types are only used internally and not in external function signatures.

- useCase- allows developer to specify the choice between v1 and v2.
- In order to call a smart contract function, we need to use the ABI (Application binary interface) specifications in order to specify the function to be called and encode the parameters, which will be included in the data field of the transaction and send it to the Ethereum network to be executed. ABI encoding is also used for events and return types, more details can be found in the documentation.
- The new ABI coder (v2) is able to encode and decode arbitrarily nested arrays and structs. Apart from supporting more types, it involves more extensive validation and safety checks, which may result in higher gas costs, but also heightened security. It is considered non-experimental as of Solidity 0.6.0 and it is enabled by default starting with Solidity 0.8.0. The old ABI coder can still be selected using pragma abicoder v1;.
- Can use pragma abicoder v2; if you are using Solidity version: 0.7.5. And for the versions below 0.7.5 we need to use the experimental version: pragma experimental ABIEncoderV2;
- An ABI allows us, i.e. our smart contracts to access the data structures and functions of another smart contract, which runs as machine code, i.e. bytecode on an EVM.

Example 1 – Working with ABI Coder v2 Pragma
In the below example, we are creating contract with struct types which is not supported by default in the solidity version 0.7.0 hence it throws a compilation error.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;
contract Test {
struct S { uint a; uint[] b; T[] c; }
struct T { uint x; uint y; }
function f(S memory, T memory, uint) public pure {}
```

```
function g() public pure returns (S memory, T memory, uint) {}
}
```

Compilation error –

```
contracts/Sample.sol:10:12: TypeError: This type is only supported in ABI coder v2.
Use "pragma abicoder v2;" to enable the feature.
function f(S memory, T memory, uint) public pure {}
^------^
```

To avoid this issue there are two options. First, we can use pragma abicoder v2; directive to make the code compatible with ABI Coder v2 that supports struct. We can see below that there is no compilation error this time.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.7.0;
pragma abicoder v2;
contract Test {
struct S { uint a; uint[] b; T[] c; }
struct T { uint x; uint y; }
function f(S memory, T memory, uint) public pure {}
function g() public pure returns (S memory, T memory, uint) {}
}
```

Another approach is to use Solidity 0.8.0 as it uses v2 ABI coder by default as shown below.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract Test {
struct S { uint a; uint[] b; T[] c; }
struct T { uint x; uint y; }
function f(S memory, T memory, uint) public pure {}
function g() public pure returns (S memory, T memory, uint) {}
}
```

Example 2 – Working with ABI Coder v1 Pragma

Since in 0.8.0 the v2 ABI Coder is used by default, and if you wish to use v1 then you can do it with pragma abicoder v1 as shown below.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
pragma abicoder v1;
contract DemoContract
{
  uint sampleData;
```

```
  function set(uint a) public
  {
    sampleData = a;
  }

  function getData() public view returns (uint)
  {
    return sampleData;
  }
}
```

\\\
\\\\\\\

7. Experimental Pragma: This can be used to enable features of the compiler or language that are not yet enabled by default
    a. SMTChecker: The use of “pragma experimental SMTChecker;” performs additional safety checks which are obtained by querying an SMT solver (See SMTChecker)
    b. The SMTChecker module automatically tries to prove that the code satisfies the specification given by require and assert statements. That is, it considers require statements as assumptions and tries to prove that the conditions inside assert statements are always true. If an assertion failure is found, a counterexample may be given to the user showing how the assertion can be violated. If no warning is given by the SMTChecker for a property, it means that the property is safe.
    c. Other checks: Arithmetic underflow and overflow, Division by zero, Trivial conditions and unreachable code, Popping an empty array, Out of bounds index access, Insufficient funds for a transfer.

8. NatSpec Comments: NatSpec stands for “Ethereum Natural Language Specification Format.” These are written with a triple slash (///) or a double asterisk block(/** ... */) directly above function declarations or statements to generate documentation in JSON format for developers and end-users. It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). These comments contain different types of tags:
    a. @title: A title that should describe the contract/interface
    b. @author: The name of the author (for contract, interface)
    c. @notice: Explain to an end user what this does (for contract, interface, function, public state variable, event)
    d. @dev: Explain to a developer any extra details (for contract, interface, function, state variable, event)

e. @param: Documents a parameter (just like in doxygen) and must be followed by parameter name (for function, event)
f. @return: Documents the return variables of a contract's function (function, public state variable)
g. @inheritdoc: Copies all missing tags from the base function and must be followed by the contract name (for function, public state variable)
h. @custom…: Custom tag, semantics is application-defined (for everywhere)

9. State Visibility Specifiers: State variables have to be specified as being public, internal or private:
   a. public: Public state variables are part of the contract interface and can be either accessed internally or via messages. An automatic getter function is generated.
   b. internal: Internal state variables can only be accessed internally from within the current contract or contracts deriving from it
   c. private: Private state variables can only be accessed from the contract they are defined in and not even in derived contracts. Everything that is inside a contract is visible to all observers external to the blockchain. Making variables private only prevents other contracts from reading or modifying the information, but it will still be visible to the whole world outside of the blockchain.
   d. State Variable cannot be marked as external.

10. State Variables: Constant & Immutable
    a. State variables can be declared as constant or immutable. In both cases, the variables cannot be modified after the contract has been constructed. For constant variables, the value has to be fixed at compile-time, while for immutable, it can still be assigned at construction time i.e. in the constructor or point of declaration.
    b. For constant variables, the value has to be a constant at compile time and it has to be assigned where the variable is declared. Any expression that accesses storage, blockchain data (e.g. block.timestamp, address(this).balance or block.number) or execution data (msg.value or gasleft()) or makes calls to external contracts is disallowed. Expressions that might have a side-effect on memory allocation are allowed, but those that might have a side-effect on other memory objects are not. The built-in functions keccak256, sha256, ripemd160, ecrecover, addmod and mulmod are allowed (even though, with the exception of keccak256, they do call external contracts).
    c. The reason behind allowing side-effects on the memory allocator is that it should be possible to construct complex objects like e.g. lookup-tables. This feature is not yet fully usable.
    d. Variables declared as immutable are a bit less restricted than those declared as constant. Immutable variables can be assigned an arbitrary value in the constructor of the contract or at the point of their declaration. They cannot be read during construction time and can only be assigned once.
    e. The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

f. The contract creation code generated by the compiler will modify the contract's runtime code before it is returned by replacing all references to immutables with the values assigned to them. This is important if you are comparing the runtime code generated by the compiler with the one actually stored in the blockchain.
g. Immutables that are assigned at their declaration are only considered initialized once the constructor of the contract is executing. This means you cannot initialize immutables inline with a value that depends on another immutable. You can do this, however, inside the constructor of the contract.
h. This is a safeguard against different interpretations about the order of state variable initialization and constructor execution, especially with regards to inheritance.

11. Compared to regular state variables, the gas costs of constant and immutable variables are much lower:
    a. For a constant variable, the expression assigned to it is copied to all the places where it is accessed and also re-evaluated each time. This allows for local optimizations.
    b. It is also possible to define constant variables at the file level.
    c. Immutable variables are evaluated once at construction time and their value is copied to all the places in the code where they are accessed. For these values, 32 bytes are reserved, even if they would fit in fewer bytes. Due to this, **constant values can sometimes be cheaper than immutable values.**
    d. Not all types for constants and immutables are implemented at this time. The only supported types are strings (only for constants) and value types.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.4;

uint constant X = 32**22 + 8;

contract C {
    string constant TEXT = "abc";
    bytes32 constant MY_HASH = keccak256("abc");
    uint immutable decimals;
    uint immutable maxBalance;
    address immutable owner = msg.sender;

    constructor(uint decimals_, address ref) {
        decimals = decimals_;
        // Assignments to immutables can even access the environment.
        maxBalance = ref.balance;
    }

    function isBalanceTooHigh(address other) public view returns
(bool) {
        return other.balance > maxBalance;
    }
}
```

12. Functions: Functions are the executable units of code. Functions are usually defined inside a contract, but they can also be defined outside of contracts. They have different levels of visibility towards other contracts. Functions outside of a contract, also called “free functions”, always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.

13. Function Visibility Specifiers: Functions have to be specified as being public, external, internal or private:
    a. public: Public functions are part of the contract interface and can be either called internally or via messages.
    b. external: External functions are part of the contract interface, which means they can be called from other contracts and via transactions. An external function f cannot be called internally (i.e. f() does not work, but this.f() works).
    c. internal: Internal functions can only be accessed internally from within the current contract or contracts deriving from it
    d. private: Private functions can only be accessed from the contract they are defined in and not even in derived contracts

14. Function Mutability Specifiers: Functions can be specified as being pure or view:
    a. view functions can read contract state but cannot modify it. This is enforced at runtime via STATICCALL opcode. The following are considered state modifying: 1) Writing to state variables 2) Emitting events 3) Creating other contracts 4) Using selfdestruct 5) Sending Ether via calls 6) Calling any function not marked view or pure 7) Using low-level calls 8) Using inline assembly that contains certain opcodes.
    b. pure functions can neither read contract state nor modify it. The following are considered reading from state: 1) Reading from state variables 2) Accessing address(this).balance or <address>.balance 3) Accessing any of the members of block, tx, msg (with the exception of msg.sig and msg.data) 4) Calling any function not marked pure 5) Using inline assembly that contains certain opcodes.
    c. It is not possible to prevent functions from reading the state at the level of the EVM. It is only possible to prevent them from writing to the state via STATICCALL. Therefore, only view can be enforced at the EVM level, but not pure.

15. Function Overloading: A contract can have multiple functions of the same name but with different parameter types. This process is called “overloading.”
    a. Overloaded functions are selected by matching the function declarations in the current scope to the arguments supplied in the function call.
    b. Return parameters are not taken into account for overload resolution.

16. Events: They are an abstraction on top of the EVM’s logging functionality. Emitting events cause the arguments to be stored in the transaction’s log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is

accessible. The Log and its event data is not accessible from within contracts (not even from the contract that created them). Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

17. Indexed Event Parameters: Adding the attribute indexed for up to three parameters adds them to a special data structure known as “topics” instead of the data part of the log. If you use arrays (including string and bytes) as indexed arguments, its Keccak-256 hash is stored as a topic instead, this is because a topic can only hold a single word (32 bytes). All parameters without the indexed attribute are ABI-encoded into the data part of the log. Topics allow you to search for events, for example when filtering a sequence of blocks for certain events. You can also filter events by the address of the contract that emitted the event.

18. Constructor: When a contract is created, its constructor (a function declared with the constructor keyword) is executed once. A constructor is optional and only one constructor is allowed. After the constructor has executed, the final code of the contract is stored on the blockchain. This code includes all public and external functions and all functions that are reachable from there through function calls. The deployed code does not include the constructor code or internal functions only called from the constructor.

19. Receive Function: A contract can have at most one receive function, declared using receive() external payable { ... } without the function keyword. This function cannot have arguments, cannot return anything and must have external visibility and payable state mutability.
    a. The receive function is executed on a call to the contract with empty calldata. This is the function that is executed on plain Ether transfers via .send() or .transfer().
    b. In the worst case, the receive function can only rely on 2300 gas being available (for example when send or transfer is used), leaving little room to perform other operations except basic logging
    c. A contract without a receive Ether function can receive Ether as a recipient of a coinbase transaction (aka miner block reward) or as a destination of a selfdestruct. A contract cannot react to such Ether transfers and thus also cannot reject them. This means that address(this).balance can be higher than the sum of some manual accounting implemented in a contract (i.e. having a counter updated in the receive Ether function).

20. Fallback Function: A contract can have at most one fallback function, declared using either fallback () external [payable] or fallback (bytes calldata _input) external [payable] returns (bytes memory _output), both without the function keyword. This function must have external visibility.
    a. The fallback function is executed on a call to the contract if none of the other functions match the given function signature, or if no data was supplied at all and there is no receive Ether function. The fallback function always receives data, but in order to also receive Ether it must be marked payable.

b. In the worst case, if a payable fallback function is also used in place of a receive function, it can only rely on 2300 gas being available

21. Solidity is a statically-typed language, which means that the type of each variable (state and local) needs to be specified in code at compile-time. This is unlike dynamically-typed languages where types are required only with runtime values. Statically-typed languages perform compile-time type-checking according to the language rules.

22. Solidity has two categories of types: Value Types and Reference Types. Value Types are called so because variables of these types will always be passed by value, i.e. they are always copied when they are used as function arguments or in assignments. In contrast, Reference Types can be modified through multiple different names i.e. references to the same underlying variable.

23. Value Types: Types that are passed by value, i.e. they are always copied when they are used as function arguments or in assignments — Booleans, Integers, Fixed Point Numbers, Address, Contract, Fixed-size Byte Arrays (bytes1, bytes2, …, bytes32), Literals (Address, Rational, Integer, String, Unicode, Hexadecimal), Enums, Functions.

24. Reference Types: Types that can be modified through multiple different names. Arrays (including Dynamically-sized bytes array bytes and string), Structs, Mappings.

25. Default Values: A variable which is declared will have an initial default value whose byte-representation is all zeros. The “default values” of variables are the typical “zero-state” of whatever the type is. For example, the default value for a bool is false. The default value for the uint or int types is 0. For statically-sized arrays and bytes1 to bytes32, each individual element will be initialized to the default value corresponding to its type. For dynamically-sized arrays, bytes and string, the default value is an empty array or string. For the enum type, the default value is its first member.

26. Scoping: Scoping in Solidity follows the widespread scoping rules of C99
    a. Variables and other items declared outside of a code block, for example functions, contracts, user-defined types, etc., are visible even before they were declared. This means you can use state variables before they are declared and call functions recursively.

27. Integers: int / uint: Signed and unsigned integers of various sizes. Keywords uint8 to uint256 in steps of 8 (unsigned of 8 up to 256 bits) and int8 to int256. uint and int are aliases for uint256 and int256, respectively. Operators are:
    a. Comparisons: <=, <, ==, !=, >=, > (evaluate to bool)
    b. Bit operators: &, |, ^ (bitwise exclusive or), ~ (bitwise negation)
    c. Shift operators: << (left shift), >> (right shift)

d. Arithmetic operators: +, -, unary - (only for signed integers), *, /, % (modulo), ** (exponentiation)

28. Integers in Solidity are restricted to a certain range. For example, with uint32, this is 0 up to 2**32 - 1. There are two modes in which arithmetic is performed on these types: The “wrapping” or “unchecked” mode and the “checked” mode. By default, arithmetic is always “checked”, which means that if the result of an operation falls outside the value range of the type, the call is reverted through a failing assertion. You can switch to “unchecked” mode using unchecked { ... }. This was introduced in compiler version 0.8.0.

29. Fixed Point Numbers: Fixed point numbers using keywords fixed / ufixed are not fully supported by Solidity yet. They can be declared, but cannot be assigned to or from. There are fixed-point libraries that are widely used for this such as DSMath, PRBMath, ABDKMath64x64 etc.
    A. From other source:-
    There are two types of fixed point numbers:

    - fixed - signed fixed point number.
    - ufixed - unsigned fixed point number.

    This value type also can be declared keywords such as ufixedMxN and fixedMxN. The M represents the amount of bits that the type takes, with N representing the number of decimal points that are available. M has to be divisible by 8, and a number from 8 to 256. N has to be a value between 0 and 80, also being inclusive.

    Note: fixed point numbers can be declared in Solidity, but they are not completely supported by this language.

30. Address Type: The address type comes in two types: (1) address: Holds a 20 byte value (size of an Ethereum address) (2) address payable: Same as address, but with the additional members transfer and send. address payable is an address you can send Ether to, while a plain address cannot be sent Ether.
    a. Operators are <=, <, ==, !=, >= and >
    b. Conversions: Implicit conversions from address payable to address are allowed, whereas conversions from address to address payable must be explicit via payable(<address>). Explicit conversions to and from address are allowed for uint160, integer literals, bytes20 and contract types.
    c. Only expressions of type address and contract-type can be converted to the type address payable via the explicit conversion payable(...). For contract-type, this conversion is only allowed if the contract can receive Ether, i.e., the contract either has a receive or a payable fallback function.

Implicit conversions of addresses:

- From address payable to address: allowed.
- From address to address payable: not allowed. This type of conversion is only possible with intermediate conversion to uint160.
- Address literals can be converted to address payable.

Explicit conversions to and from address are permitted for integers, integer literals, contact types and bytes20. However, Solidity prevents the conversions of address payable(x).

The address(x) can be converted to address payable in cases when x is of integer, fixed bytes type, or a literal or a contract that has a payable fallback function. When x is a contract without the payable fallback function, the address(x) is of type address.

Note: it is possible to disregard the difference between address and address payable by using the address. You can call the transfer function on msg.sender which is an address payable.

31. Members of Address Type:
    a. <address>.balance (uint256): balance of the Address in Wei
    b. <address>.code (bytes memory): code at the Address (can be empty)
    c. <address>.codehash (bytes32): the codehash of the Address
    d. <address payable>.transfer(uint256 amount): send given amount of Wei to Address, reverts on failure, forwards 2300 gas stipend, not adjustable
    e. <address payable>.send(uint256 amount) returns (bool): send given amount of Wei to Address, returns false on failure, forwards 2300 gas stipend, not adjustable
    f. <address>.call(bytes memory) returns (bool, bytes memory): issue low-level CALL with the given payload, returns success condition and return data, forwards all available gas, adjustable
    g. <address>.delegatecall(bytes memory) returns (bool, bytes memory): issue low-level DELEGATECALL with the given payload, returns success condition and return data, forwards all available gas, adjustable
    h. <address>.staticcall(bytes memory) returns (bool, bytes memory): issue low-level STATICCALL with the given payload, returns success condition and return data, forwards all available gas, adjustable

Members of Address

Example
```
address payable x = address(0x123);
address myAddress = address(this);
if (x.balance < 10 && myAddress.balance >= 10) x.transfer(10);
```

When the balance of current contracts is not sufficient enough or when the receiver rejects the transfer, the transfer function fails. It reverts on failure.

Remember: the send function is not a safe alternative for transfer. The send fails to transfer when the call stack depth reaches 1024 or when the recipient no longer has gas.

32. Transfer: The transfer function fails if the balance of the current contract is not large enough or if the Ether transfer is rejected by the receiving account. The transfer function reverts on failure. The code in receive function or if not present then in fallback function is executed with the transfer call. If that execution runs out of gas or fails in any way, the Ether transfer will be reverted and the current contract will stop with an exception.

33. Send: The send function is the low-level counterpart of transfer. If the execution fails then send only returns false and does not revert unlike transfer. So the return value of send must be checked by the caller.

34. Call/Delegatecall/Staticcall: In order to interface with contracts that do not adhere to the ABI, or to get more direct control over the encoding, the functions call, delegatecall and staticcall are provided. They all take a single bytes memory parameter and return the success condition (as a bool) and the returned data (bytes memory). The functions abi.encode, abi.encodePacked, abi.encodeWithSelector and abi.encodeWithSignature can be used to encode structured data.
    a. gas and value modifiers can be used with these functions (delegatecall doesn't support value) to specify the amount of gas and Ether value passed to the callee.
    b. With delegatecall, only the code of the given address is used but all other aspects (storage, balance, msg.sender etc.) are taken from the current contract. The purpose of delegatecall is to use library/logic code which is stored in callee contract but operate on the state of the caller contract
    c. With staticcall, the execution will revert if the called function modifies the state in any way

Gain more direct control over encoding or interface with contracts (not adhere to the ABI) with call, delegatecall and staticcall functions.

They accept one bytes memory parameter, deliver the success condition as a boolean and the returned data. For encoding of structured data, use abi.encode, abi.encodePacked, abi.encodeWithSelector and abi.encodeWithSignature:

Example
bytes memory payload = abi.encodeWithSignature("register(string)", "MyName");
(bool success, bytes memory returnData) = address(nameReg).call(payload);
require(success);
You can adjust the provided gas using the .gas() modifier:

Example

address(nameReg).call.gas(1000000)(abi.encodeWithSignature("register(string)", "MyName"));
You can manipulate the Ether value as well:

Example
address(nameReg).call.value(1 ether)(abi.encodeWithSignature("register(string)", "MyName"));
It is possible to combine these modifiers. Their order is not important:

Example
address(nameReg).call.gas(1000000).value(1 ether)
(abi.encodeWithSignature("register(string)", “MyName"));

35. Contract Type: Every contract defines its own type. Contracts can be explicitly converted to and from the address type. Contract types do not support any operators. The members of contract types are the external functions of the contract including any state variables marked as public.

36. Fixed-size Byte Arrays: The value types bytes1, bytes2, bytes3, …, bytes32 hold a sequence of bytes from one to up to 32. The type byte[] is an array of bytes, but due to padding rules, it wastes 31 bytes of space for each element (except in storage). It is better to use the bytes type instead.

37. Literals: They can be of 5 types:
    a. Address Literals: Hexadecimal literals that pass the address checksum test are of address type. Hexadecimal literals that are between 39 and 41 digits long and do not pass the checksum test produce an error. The mixed-case address checksum format is defined in EIP-55.

    b. Rational and Integer Literals: Integer literals are formed from a sequence of numbers in the range 0-9. Decimal fraction literals are formed by a . with at least one number on one side. Scientific notation is also supported, where the base can have fractions and the exponent cannot. Underscores can be used to separate the digits of a numeric literal to aid readability and are semantically ignored.

    c. String Literals: String literals are written with either double or single-quotes ("foo" or ‘bar’). They can only contain printable ASCII characters and a set of escape characters

    d. Unicode Literals: Unicode literals prefixed with the keyword unicode can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

    e. Hexadecimal Literals: Hexadecimal literals are hexadecimal digits prefixed with the keyword hex and are enclosed in double or single-quotes e.g. hex” 001122FF”, hex'0011_22_FF'.

From oreilly

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.

Address Literals

How to define a variable of address type?
To define a variable of address type, specify the keyword address in front of the variable name.

address user = msg.sender
We have used the Solidity built-in function msg.sender to retrieve the address of the current account interacting with the smart contract.

But you can also hardcode specific addresses in your Solidity code, using address literals. These are described in the next section.

Address literals
Address literals are the hexadecimal representation of an Ethereum address, hardcoded in a solidity file.

Here is an example of how to declare an address literals in Solidity .

address owner = 0xc0ffee254729296a45a3885639AC7E10F9d54979
As seen before, an address literal must:

contain 40 character (20 bytes long), and
be prefixed with 0x.
have a valid checksum
Regarding 3), address literals must have a valid checksum. If they do not pass the checksum test, Remix or the Solidity compiler will bring up a warning and will treat it as a regular rational number intervals.

The mixed-case address checksum format is defined in EIP-55

The website ethsum.netlify.com is a really useful tool that enables you to convert an ethereum address (in all lower case) into a checksumed address.

Finally, address literals are set as address payable by default.

Addresses
The address Solidity value type has two similar kinds:

address holds a 20-byte value (size of an Ethereum address).
address payable is the same as address, but have transfer and send members.
Note: there are two distinctions between smart contract addresses - the address payable can receive Ether, while simple address cannot.

Implicit conversions of addresses:

From address payable to address: allowed.
From address to address payable: not allowed. This type of conversion is only possible with intermediate conversion to uint160.
Address literals can be converted to address payable.

Explicit conversions to and from address are permitted for integers, integer literals, contact types and bytes20. However, Solidity prevents the conversions of address payable(x).

The address(x) can be converted to address payable in cases when x is of integer, fixed bytes type, or a literal or a contract that has a payable fallback function. When x is a contract without the payable fallback function, the address(x) is of type address.

Note: it is possible to disregard the difference between address and address payable by using the address. You can call the transfer function on msg.sender which is an address payable.

The type returned by msg.sender is of address payable type

Integer literals
From docs

Rational and Integer Literals

Integer literals are formed from a sequence of digits in the range 0-9. They are interpreted as decimals. For example, 69 means sixty nine. Octal (using a system of numerical notation that has 8 rather than 10 as a base.) literals do not exist in Solidity and leading zeros are invalid.

Decimal fractional literals are formed by a . with at least one number after the decimal point. Examples include .1 and 1.3 (but not 1.).

Scientific notation in the form of 2e10 is also supported, where the mantissa can be fractional but the exponent has to be an integer. The literal MeE is equivalent to M * 10**E. Examples include 2e10, -2e10, 2e-10, 2.5e1.

Underscores can be used to separate the digits of a numeric literal to aid readability. For example, decimal 123_000, hexadecimal 0x2eff_abde, scientific decimal notation 1_2e345_678 are all valid. Underscores are only allowed between two digits and only one consecutive underscore is allowed. There is no additional semantic meaning added to a number literal containing underscores, the underscores are ignored.

Number literal expressions retain arbitrary precision until they are converted to a non-literal type (i.e. by using them together with anything other than a number literal expression (like boolean literals) or by explicit conversion). This means that computations do not overflow and divisions do not truncate in number literal expressions.

For example, (2**800 + 1) - 2**800 results in the constant 1 (of type uint8) although intermediate results would not even fit the machine word size. Furthermore, .5 * 8 results in the integer 4 (although non-integers were used in between).

Warning

While most operators produce a literal expression when applied to literals, there are certain operators that do not follow this pattern:

Ternary operator (... ? ... : ...),

Array subscript (<array>[<index>]).

You might expect expressions like 255 + (true ? 1 : 0) or 255 + [1, 2, 3][0] to be equivalent to using the literal 256 directly, but in fact they are computed within the type uint8 and can overflow.

String Literals

String literals are written with either double or single-quotes ("foo" or 'bar'), and they can also be split into multiple consecutive parts ("foo" "bar" is equivalent to "foobar") which can be helpful when dealing with long strings. They do not imply trailing zeroes as in C; "foo" represents three bytes, not four. As with integer literals, their type can vary, but they are implicitly convertible to bytes1, ..., bytes32, if they fit, to bytes and to string.

For example, with bytes32 samevar = "stringliteral" the string literal is interpreted in its raw byte form when assigned to a bytes32 type.

String literals can only contain printable ASCII characters, which means the characters between and including 0x20 .. 0x7E.

Additionally, string literals also support the following escape characters:

\<newline> (escapes an actual newline)

\\ (backslash)

\' (single quote)

\" (double quote)

\n (newline)

\r (carriage return)

\t (tab)

\xNN (hex escape, see below)

\uNNNN (unicode escape, see below)

\xNN takes a hex value and inserts the appropriate byte, while \uNNNN takes a Unicode codepoint and inserts an UTF-8 sequence.

Note

Until version 0.8.0 there were three additional escape sequences: \b, \f and \v. They are commonly available in other languages but rarely needed in practice. If you do need them, they can still be inserted via hexadecimal escapes, i.e. \x08, \x0c and \x0b, respectively, just as any other ASCII character.

The string in the following example has a length of ten bytes. It starts with a newline byte, followed by a double quote, a single quote a backslash character and then (without separator) the character sequence abcdef.

```
"\n\"\'\\abc\
def"
```

Any Unicode line terminator which is not a newline (i.e. LF, VF, FF, CR, NEL, LS, PS) is considered to terminate the string literal. Newline only terminates the string literal if it is not preceded by a \.

Unicode Literals

While regular string literals can only contain ASCII, Unicode literals – prefixed with the keyword unicode – can contain any valid UTF-8 sequence. They also support the very same escape sequences as regular string literals.

string memory a = unicode"Hello 😃";

Hexadecimal literals

Hexadecimal literals are written as ordinary string literals, enclosed in single or double quotes, but are prefixed with a keyword hex, e.g. hex"001122FF", hex'0011_22_FF'.

Multiple hexadecimal literals separated by whitespace are concatenated into a single literal: hex"00112233" hex"44556677" is equivalent to hex"0011223344556677"

Hexadecimal literals behave like string literals and have the same convertibility restrictions.

The content of a hexadecimal literal is represented by hexadecimal digits, which can optionally use one underscore as a separator between byte boundaries. The value of the hexadecimal literal will be the binary representation of the hexadecimal sequence:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;
contract hexadecimal_literal {
    function get_literal()
    public
    pure
    returns (
        bytes memory
    )
    {
       return hex'0011_22_FF';
    }
}
```

The result of calling the get_literal() function is:
bytes: 0x001122FF

In a case of multiple hexadecimal literals separated by a whitespace character, the result is a single concatenated literal, e.g.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;
contract hexadecimal_literal {
    function get_literal()
```

```
    public
    pure
    returns (
        bytes memory
    )
    {
      return hex"00112233" hex"44556677";
    }
}
```

The result of calling the get_literal() function is:
bytes: 0x0011223344556677.

Owing to their similarity, hexadecimal literals share the same convertibility restrictions as string literals.

\\\
\\\\\

38. Enums: Enums are one way to create a user-defined type in Solidity. They are explicitly convertible to and from all integer types but implicit conversion is not allowed. The explicit conversion from integer checks at runtime that the value lies inside the range of the enum and causes a Panic error otherwise. They require at least one member and its default value when declared is the first member. They cannot have more than 256 members.

Using type(NameOfEnum).min and type(NameOfEnum).max you can get the smallest and respectively largest value of the given enum.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.8;

contract test {
    enum ActionChoices { GoLeft, GoRight, GoStraight, SitStill }
    ActionChoices choice;
    ActionChoices constant defaultChoice = ActionChoices.GoStraight;

    function setGoStraight() public {
        choice = ActionChoices.GoStraight;
    }

    // Since enum types are not part of the ABI, the signature of 
"getChoice"
    // will automatically be changed to "getChoice() returns (uint8)"
    // for all matters external to Solidity.
    function getChoice() public view returns (ActionChoices) {
        return choice;
    }
```

```
    function getDefaultChoice() public pure returns (uint) {
        return uint(defaultChoice);
    }

    function getLargestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).max;
    }

    function getSmallestValue() public pure returns (ActionChoices) {
        return type(ActionChoices).min;
    }
}
```

Note : Enums can also be declared on the file level, outside of contract or library definitions.

39. Function Types: Function types are the types of functions. Variables of function type can be assigned from functions and function parameters of function type can be used to pass functions to and return functions from function calls. They come in two flavours - internal and external functions. Internal functions can only be called inside the current contract. External functions consist of an address and a function signature and they can be passed via and returned from external function calls.

By default, function types are internal, so the internal keyword can be omitted. Note that this only applies to function types. Visibility has to be specified explicitly for functions defined in contracts, they do not have a default.

A function type `A` is implicitly convertible to a function type `B` if and only if their parameter types are identical, their return types are identical, their internal/external property is identical and the state mutability of `A` is more restrictive than the state mutability of `B`. In particular:

- `pure` functions can be converted to `view` and `non-payable` functions
- `view` functions can be converted to `non-payable` functions
- `payable` functions can be converted to `non-payable` functions

More information can be found in documentation.

40. Reference Types & Data Location: Every reference type has an additional annotation — the data location where it is stored. There are three data locations: memory, storage and calldata.
    a. memory: whose lifetime is limited to an external function call
    b. storage: whose lifetime is limited to the lifetime of a contract and the location where the state variables are stored
    c. calldata: which is a non-modifiable, non-persistent area where function arguments are stored and behaves mostly like memory. It is required for parameters of external functions but can also be used for other variables.

Values of reference type can be modified through multiple different names. Contrast this with value types where you get an independent copy whenever a variable of value type is used. Because of that, reference types have to be handled more carefully than value types. Currently, reference types comprise structs, arrays and mappings. If you use a reference type, you always have to explicitly provide the data area where the type is stored: memory (whose lifetime is limited to an external function call), storage (the location where the state variables are stored, where the lifetime is limited to the lifetime of a contract) or calldata (special data location that contains the function arguments).

An assignment or type conversion that changes the data location will always incur an automatic copy operation, while assignments inside the same data location only copy in some cases for storage types.

41. Data Location & Assignment: Data locations are not only relevant for persistence of data, but also for the semantics of assignments.
    a. Assignments between storage and memory (or from calldata) always create an independent copy.

    b. Assignments from memory to memory only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.

    c. Assignments from storage to a local storage variable also only assign a reference.

    d. All other assignments to storage always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

Data location
Every reference type has an additional annotation, the “data location”, about where it is stored. There are three data locations: memory, storage and calldata. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

Note : If you can, try to use calldata as data location because it will avoid copies and also makes sure that the data cannot be modified. Arrays and structs with calldata data location can also be returned from functions, but it is not possible to allocate such types.

Note : Prior to version 0.6.9 data location for reference-type arguments was limited to calldata in external functions, memory in public functions and either memory or storage in internal and private ones. Now memory and calldata are allowed in all functions regardless of their visibility.

Note : Prior to version 0.5.0 the data location could be omitted, and would default to different locations depending on the kind of variable, function type, etc., but all complex types must now give an explicit data location.

Data location and assignment behaviour
Data locations are not only relevant for persistency of data, but also for the semantics of assignments:

- Assignments between storage and memory (or from calldata) always create an independent copy.

- Assignments from memory to memory only create references. This means that changes to one memory variable are also visible in all other memory variables that refer to the same data.

- Assignments from storage to a local storage variable also only assign a reference.

- All other assignments to storage always copy. Examples for this case are assignments to state variables or to members of local variables of storage struct type, even if the local variable itself is just a reference.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;

contract C {
    // The data location of x is storage.
    // This is the only place where the
    // data location can be omitted.
    uint[] x;

    // The data location of memoryArray is memory.
    function f(uint[] memory memoryArray) public {
        x = memoryArray; // works, copies the whole array to storage
        uint[] storage y = x; // works, assigns a pointer, data
location of y is storage
        y[7]; // fine, returns the 8th element
        y.pop(); // fine, modifies x through y
        delete x; // fine, clears the array, also modifies y
        // The following does not work; it would need to create a new
temporary /
        // unnamed array in storage, but storage is "statically"
allocated:
        // y = memoryArray;
        // Similarly, "delete y" is not valid, as assignments to local
variables
        // referencing storage objects can only be made from existing
storage objects.
        // It would "reset" the pointer, but there is no sensible
location it could point to.
```

```
        // For more details see the documentation of the "delete"
operator.
        // delete y;
        g(x); // calls g, handing over a reference to x
        h(x); // calls h and creates an independent, temporary copy in
memory
    }

    function g(uint[] storage) internal pure {}
    function h(uint[] memory) public pure {}
}
```

42. Arrays: Arrays can have a compile-time fixed size, or they can have a dynamic size
    a. The type of an array of fixed size k and element type T is written as T[k], and an array of dynamic size as T[].
    b. Indices are zero-based
    c. Array elements can be of any type, including mapping or struct.
    d. Accessing an array past its end causes a failing assertion

43. Array members:
    a. length: returns number of elements in array
    b. push(): appends a zero-initialised element at the end of the array and returns a reference to the element
    c. push(x): appends a given element at the end of the array and returns nothing
    d. pop: removes an element from the end of the array and implicitly calls delete on the removed element

44. Variables of type bytes and string are special arrays
    a. bytes is similar to byte[], but it is packed tightly in calldata and memory
    b. string is equal to bytes but does not allow length or index access
    c. Solidity does not have string manipulation functions, but there are third-party string libraries
    d. Use bytes for arbitrary-length raw byte data and string for arbitrary-length string (UTF-8) data
    e. Use bytes over byte[] because it is cheaper, since byte[] adds 31 padding bytes between the elements
    f. If you can limit the length to a certain number of bytes, always use one of the value types bytes1 to bytes32 because they are much cheaper

45. Memory Arrays: Memory arrays with dynamic length can be created using the new operator
    a. As opposed to storage arrays, it is not possible to resize memory arrays i.e. the .push member functions are not available

b. You either have to calculate the required size in advance or create a new memory array and copy every element

As all variables in Solidity, the elements of newly allocated arrays are always initialized with the default value.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.16 <0.9.0;

contract C {
    function f(uint len) public pure {
        uint[] memory a = new uint[](7);
        bytes memory b = new bytes(len);
        assert(a.length == 7);
        assert(b.length == len);
        a[6] = 8;
    }
}
```

46. Array Literals: An array literal is a comma-separated list of one or more expressions, enclosed in square brackets ([…])
    a. It is always a statically-sized memory array whose length is the number of expressions
    b. The base type of the array is the type of the first expression on the list such that all other expressions can be implicitly converted to it. It is a type error if this is not possible.
    c. Fixed size memory arrays cannot be assigned to dynamically-sized memory arrays

47. Gas costs of push and pop: Increasing the length of a storage array by calling push() has constant gas costs because storage is zero-initialised, while decreasing the length by calling pop() has a cost that depends on the “size” of the element being removed. If that element is an array, it can be very costly, because it includes explicitly clearing the removed elements similar to calling delete on them.

48. Array Slices: Array slices are a view on a contiguous portion of an array. They are written as x[start:end], where start and end are expressions resulting in a uint256 type (or implicitly convertible to it). The first element of the slice is x[start] and the last element is x[end - 1]
    a. If start is greater than end or if end is greater than the length of the array, an exception is thrown
    b. Both start and end are optional: start defaults to 0 and end defaults to the length of the array
    c. Array slices do not have any members
    d. They are implicitly convertible to arrays of their underlying type and support index access. Index access is not absolute in the underlying array, but relative to the start of the slice

e. Array slices do not have a type name which means no variable can have an array slices as type and they only exist in intermediate expressions
f. Array slices are only implemented for calldata arrays.
g. Array slices are useful to ABI-decode secondary data passed in function parameters

49. Struct Types: Structs help define new aggregate types by combining other value/reference types into one unit. Struct types can be used inside mappings and arrays and they can themselves contain mappings and arrays. It is not possible for a struct to contain a member of its own type

50. Mapping Types: Mappings define key-value pairs and are declared using the syntax mapping(_KeyType => _ValueType) _VariableName.
    a. The _KeyType can be any built-in value type, bytes, string, or any contract or enum type. Other user-defined or complex types, such as mappings, structs or array types are not allowed. _ValueType can be any type, including mappings, arrays and structs.
    b. Key data is not stored in a mapping, only its keccak256 hash is used to look up the value
    c. They do not have a length or a concept of a key or value being set
    d. They can only have a data location of storage and thus are allowed for state variables, as storage reference types in functions, or as parameters for library functions
    e. They cannot be used as parameters or return parameters of contract functions that are publicly visible. These restrictions are also true for arrays and structs that contain mappings.
    f. You cannot iterate over mappings, i.e. you cannot enumerate their keys. It is possible, though, to implement a data structure on top of them and iterate over that.

51. Operators Involving LValues (i.e. a variable or something that can be assigned to)
    a. a += e is equivalent to a = a + e. The operators -=, *=, /=, %=, |=, &= and ^= are defined accordingly
    b. a++ and a-- are equivalent to a += 1 / a -= 1 but the expression itself still has the previous value of a
    c. In contrast, --a and ++a have the same effect on a but return the value after the change

52. delete
    a. delete a assigns the initial value for the type to a
    b. For integers it is equivalent to a = 0
    c. For arrays, it assigns a dynamic array of length zero or a static array of the same length with all elements set to their initial value
    d. delete a[x] deletes the item at index x of the array and leaves all other elements and the length of the array untouched
    e. For structs, it assigns a struct with all members reset

f. delete has no effect on mappings. So if you delete a struct, it will reset all members that are not mappings and also recurse into the members unless they are mappings.
g. For mappings, individual keys and what they map to can be deleted: If a is a mapping, then delete a[x] will delete the value stored at x

53. Implicit Conversions: An implicit type conversion is automatically applied by the compiler in some cases during assignments, when passing arguments to functions and when applying operators
    a. implicit conversion between value-types is possible if it makes sense semantically and no information is lost
    b. For example, uint8 is convertible to uint16 and int128 to int256, but int8 is not convertible to uint256, because uint256 cannot hold values such as -1

54. Explicit Conversions: If the compiler does not allow implicit conversion but you are confident a conversion will work, an explicit type conversion is sometimes possible. This may result in unexpected behaviour and allows you to bypass some security features of the compiler e.g. int to uint
    a. If an integer is explicitly converted to a smaller type, higher-order bits are cut off
    b. If an integer is explicitly converted to a larger type, it is padded on the left (i.e., at the higher order end)
    c. Fixed-size bytes types while explicitly converting to a smaller type and will cut off the bytes to the right
    d. Fixed-size bytes types while explicitly converting to a larger type and will pad bytes to the right.

55. Conversions between Literals and Elementary Types
    a. Decimal and hexadecimal number literals can be implicitly converted to any integer type that is large enough to represent it without truncation
    b. Decimal number literals cannot be implicitly converted to fixed-size byte arrays
    c. Hexadecimal number literals can be, but only if the number of hex digits exactly fits the size of the bytes type. As an exception both decimal and hexadecimal literals which have a value of zero can be converted to any fixed-size bytes type
    d. String literals and hex string literals can be implicitly converted to fixed-size byte arrays, if their number of characters matches the size of the bytes type

56. A literal number can take a suffix of wei, gwei (1e9) or ether (1e18) to specify a sub-denomination of Ether

57. Suffixes like seconds, minutes, hours, days and weeks after literal numbers can be used to specify units of time where seconds are the base unit where 1 == 1 seconds,1 minutes == 60 seconds, 1 hours == 60 minutes, 1 days == 24 hours and 1 weeks == 7 days

a. Take care if you perform calendar calculations using these units, because not every year equals 365 days and not even every day has 24 hours because of leap seconds
b. These suffixes cannot be applied directly to variables but can be applied by multiplication

58. Block and Transaction Properties:
    a. blockhash(uint blockNumber) returns (bytes32): hash of the given block - only works for 256 most recent, excluding current, blocks
    b. block.chainid (uint): current chain id
    c. block.coinbase (address payable): current block miner's address
    d. block.difficulty (uint): current block difficulty
    e. block.gaslimit (uint): current block gaslimit
    f. block.number (uint): current block number
    g. block.timestamp (uint): current block timestamp as seconds since unix epoch
    h. msg.data (bytes calldata): complete calldata
    i. msg.sender (address): sender of the message (current call)
    j. msg.sig (bytes4): first four bytes of the calldata (i.e. function identifier)
    k. msg.value (uint): number of wei sent with the message
    l. tx.gasprice (uint): gas price of the transaction
    m. gasleft() returns (uint256): remaining gas
    n. tx.origin (address): sender of the transaction (full call chain)

59. The values of all members of msg, including msg.sender and msg.value can change for every external function call. This includes calls to library functions.

60. Do not rely on block.timestamp or blockhash as a source of randomness. Both the timestamp and the block hash can be influenced by miners to some degree. The current block timestamp must be strictly larger than the timestamp of the last block, but the only guarantee is that it will be somewhere between the timestamps of two consecutive blocks in the canonical chain.

61. The block hashes are not available for all blocks for scalability reasons. You can only access the hashes of the most recent 256 blocks, all other values will be zero.

62. ABI Encoding and Decoding Functions:
    a. abi.decode(bytes memory encodedData, (...)) returns (...): ABI-decodes the given data, while the types are given in parentheses as second argument.
    b. abi.encode(...) returns (bytes memory): ABI-encodes the given arguments
    c. abi.encodePacked(...) returns (bytes memory): Performs packed encoding of the given arguments. Note that packed encoding can be ambiguous!
    d. abi.encodeWithSelector(bytes4 selector, ...) returns (bytes memory): ABI-encodes the given arguments starting from the second and prepends the given four-byte selector
    e. abi.encodeWithSignature(string memory signature, ...) returns (bytes memory): Equivalent to abi.encodeWithSelector(bytes4(keccak256(bytes(signature))), …)

63. Error Handling:
    a. assert(bool condition): causes a Panic error and thus state change reversion if the condition is not met - to be used for internal errors.
    b. require(bool condition): reverts if the condition is not met - to be used for errors in inputs or external components.
    c. require(bool condition, string memory message): reverts if the condition is not met - to be used for errors in inputs or external components. Also provides an error message.
    d. revert(): abort execution and revert state changes
    e. revert(string memory reason): abort execution and revert state changes, providing an explanatory string

64. Mathematical and Cryptographic Functions:
    a. addmod(uint x, uint y, uint k) returns (uint): compute (x + y) % k where the addition is performed with arbitrary precision and does not wrap around at 2**256. Assert that k != 0 starting from version 0.5.0.
    b. mulmod(uint x, uint y, uint k) returns (uint): compute (x * y) % k where the multiplication is performed with arbitrary precision and does not wrap around at 2**256. Assert that k != 0 starting from version 0.5.0.
    c. keccak256(bytes memory) returns (bytes32): compute the Keccak-256 hash of the input
    d. sha256(bytes memory) returns (bytes32): compute the SHA-256 hash of the input
    e. ripemd160(bytes memory) returns (bytes20): compute RIPEMD-160 hash of the input
    f. ecrecover(bytes32 hash, uint8 v, bytes32 r, bytes32 s) returns (address): recover the address associated with the public key from elliptic curve signature or return zero on error. The function parameters correspond to ECDSA values of the signature: r = first 32 bytes of signature, s = second 32 bytes of signature, v = final 1 byte of signature. ecrecover returns an address, and not an address payable.

65. If you use ecrecover, be aware that a valid signature can be turned into a different valid signature without requiring knowledge of the corresponding private key. This is usually not a problem unless you require signatures to be unique or use them to identify items. OpenZeppelin has a ECDSA helper library that you can use as a wrapper for ecrecover without this issue.

66. Contract Related:
    a. this (current contract's type): the current contract, explicitly convertible to Address
    b. selfdestruct(address payable recipient): Destroy the current contract, sending its funds to the given Address and end execution.

67. selfdestruct has some peculiarities: the receiving contract's receive function is not executed and the contract is only really destroyed at the end of the transaction and revert's might "undo" the destruction.