



Solidity 1011	Solidity 1012
<p data-bbox="22 127 1117 462">Solidity is a high-level language for implementing smart contracts on Ethereum (and the blockchains) targeting the EVM. Solidity was proposed in 2014 by Gavin Wood and was later developed by Ethereum’s Solidity team, led by Christian Reitwiessner, Alex Beregszaszi & others. (See here)</p> 	<p data-bbox="1117 127 2217 526">It is influenced mainly by C++, a little from Python and early-on from JavaScript. The syntax and OOP concepts are from C++. Solidity’s modifiers, multiple inheritance, C3 linearization and the super keyword are influences from Python. Function-level scoping and var keyword were JavaScript influences early-on but those have been reduced since v0.4.0. (See here)</p> 
Solidity 1013	Solidity 1014
<p data-bbox="22 909 1117 1069">Solidity is statically typed, supports inheritance, libraries and complex user-defined types. It is a fully-featured high-level language.</p>	<p data-bbox="1117 909 2217 1181">The layout of a Solidity source file can contain an arbitrary number of pragma directives, import directives and struct/enum/contract definitions. The best-practices for layout within a contract is the following order: state variables, events, modifiers, constructor and functions.</p>

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 (See [SPDX](#)).



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. Complex pragmas are also possible using ‘>’, ‘>=’, ‘<’ and ‘<=’ symbols to combine multiple versions e.g. “*pragma solidity >=0.8.0 <0.8.3;*”

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

- e. 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.”

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. 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.

- c. 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.

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.



Imports: Solidity supports import statements to help modularise your code that are similar to those available in JavaScript (from ES6 on) e.g. “*import “filename”;*”

Comments: Single-line comments (//) and multi-line comments (/*...*/) are possible. Comments are recommended as in-line documentation of what contracts, functions, variables, expressions, control and data flow are expected to do as per the implementation, and any assumptions/invariants made/needed. They help in readability and maintainability.

- f. **@param:** Documents a parameter (just like in doxygen) and must be followed by parameter name (for function, event)
- 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)

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. **@return:** Documents the return variables of a contract’s function (function, public state variable)

Contracts: They are similar to classes in object-oriented languages in that they contain persistent data in state variables and functions that can modify these variables. Contracts can inherit from other contracts

Contracts can contain declarations of State Variables, Functions, Function Modifiers, Events, Errors, Struct Types and Enum Types

State Variables: They are variables that can be accessed by all functions of the contract and whose values are permanently stored in contract storage

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.

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.
- c. The compiler does not reserve a storage slot for these variables, and every occurrence is replaced by the respective value.

- d. 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.

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.

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. 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.
- c. The only supported types are strings (only for constants) and value types.

Function parameters: Function parameters are declared the same way as variables, and the name of unused parameters can be omitted. Function parameters can be used as any other local variable and they can also be assigned to.

Function Return Variables: Function return variables are declared with the same syntax after the *returns* keyword.

- a. The names of return variables can be omitted. Return variables can be used as any other local variable and they are initialized with their default value and have that value until they are (re-)assigned.
- b. You can either explicitly assign to return variables and then leave the function as above, or you can provide return values (either a single or multiple ones) directly with the return statement
- c. If you use an early return to leave a function that has return variables, you must provide return values together with the return statement
- d. When a function has multiple return types, the statement `return (v0, v1, ..., vn)` can be used to return multiple values. The number of components must be the same as the number of return variables and their types have to match, potentially after an implicit conversion

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

Function Modifiers: They can be used to change the behaviour of functions in a declarative way. For example, you can use a modifier to automatically check a condition prior to executing the function. The function's control flow continues after the “`_`” in the preceding modifier. Multiple modifiers are applied to a function by specifying them in a whitespace-separated list and are evaluated in the order presented. The modifier can choose not to execute the function body at all and in that case the return variables are set to their default values just as if the function had an empty body. The `_` symbol can appear in the modifier multiple times. Each occurrence is replaced with the function body.

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*.

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.

Free Functions: Functions that are defined outside of contracts are called “free functions” and always have implicit internal visibility. Their code is included in all contracts that call them, similar to internal library functions.

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.

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.

Struct Types: They are custom defined types that can group several variables of same/different types together to create a custom data structure. The struct members are accessed using ‘.’ e.g.: struct s {address user; uint256 amount} where s.user and s.amount access the struct members.

Emit: Events are emitted using ‘emit’, followed by the name of the event and the arguments e.g. “*emit Deposit(msg.sender, _id, msg.value);*”

Enums: They can be used to create custom types with a finite set of constant values to improve readability. They need a minimum of one member and can have a maximum of 256. They can be explicitly converted to/from integers. The options are represented by unsigned integer values starting from 0. The default value is the first member.

Constructor: Contracts can be created “from outside” via Ethereum transactions or from within Solidity contracts. 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.

- 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).

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.

- 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()*.
- 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

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.

- 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.
- 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

Solidity 10135	Solidity 10136
<p>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. Other examples are C, C++, Java, Rust, Go, Scala.</p>	<p>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.</p>
Solidity 10137	Solidity 10138
<p>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.</p>	<p>Reference Types: Types that can be modified through multiple different names. Arrays (including Dynamically-sized bytes array <i>bytes</i> and <i>string</i>), Structs, Mappings.</p>

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.

Boolean: bool Keyword and the possible values are constants true and false.

- a. Operators are ! (logical negation) && (logical conjunction, “and”) || (logical disjunction, “or”) == (equality) and != (inequality).
- b. The operators || and && apply the common short-circuiting rules. This means that in the expression f(x) || g(y), if f(x) evaluates to true, g(y) will not be evaluated even if it may have side-effects.

Scoping: Scoping in Solidity follows the widespread scoping rules of C99

- a. Variables are visible from the point right after their declaration until the end of the smallest { }-block that contains the declaration. As an exception to this rule, variables declared in the initialization part of a for-loop are only visible until the end of the for-loop.
- b. Variables that are parameter-like (function parameters, modifier parameters, catch parameters, ...) are visible inside the code block that follows - the body of the function/modifier for a function and modifier parameter and the catch block for a catch parameter.
- c. 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.

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)

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.

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.

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.

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

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.

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.

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

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.

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.

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'*.

Enums: Enums are one way to create a user-defined type in Solidity. They require at least one member and its default value when declared is the first member. They cannot have more than 256 members.

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.

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.

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.

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

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

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

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

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

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

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.

- 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

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

- 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.

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

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

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`

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.

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`

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

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

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

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)

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.

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.

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.

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))), ...)`

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

- 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.

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 $k \neq 0$ 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 $k \neq 0$ 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

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.

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.

Type Information: The expression `type(X)` can be used to retrieve information about the type `X`, where `X` can be either a contract or an integer type. For a contract type `C`, the following type information is available:

- a. *type(C).name*: The name of the contract.
- b. *type(C).creationCode*: Memory byte array that contains the creation bytecode of the contract. This can be used in inline assembly to build custom creation routines, especially by using the `create2` opcode. This property cannot be accessed in the contract itself or any derived contract. It causes the bytecode to be included in the bytecode of the call site and thus circular references like that are not possible.

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.

- c. *type(C).runtimeCode*: Memory byte array that contains the runtime bytecode of the contract. This is the code that is usually deployed by the constructor of `C`. If `C` has a constructor that uses inline assembly, this might be different from the actually deployed bytecode. Also note that libraries modify their runtime bytecode at time of deployment to guard against regular calls. The same restrictions as with *creationCode* also apply for this property.
- d. For an interface type `I`, the following type information is available:
 - type(I).interfaceId*: A bytes4 value containing the EIP-165 interface identifier of the given interface `I`. This identifier is defined as the XOR of all function selectors defined within the interface itself - excluding all inherited functions.

For an integer type T , the following type information is available:

- a. $type(T).min$: The smallest value representable by type T .
- b. $type(T).max$: The largest value representable by type T .

Exceptions: Solidity uses state-reverting exceptions to handle errors. Such an exception undoes all changes made to the state in the current call (and all its sub-calls) and flags an error to the caller

- a. When exceptions happen in a sub-call, they “bubble up” (i.e., exceptions are rethrown) automatically. Exceptions to this rule are `send` and the low-level functions `call`, `delegatecall` and `staticcall`: they return false as their first return value in case of an exception instead of “bubbling up”.
- b. Exceptions in external calls can be caught with the `try/catch` statement
- c. Exceptions can contain data that is passed back to the caller. This data consists of a 4-byte selector and subsequent ABI-encoded data. The selector is computed in the same way as a function selector, i.e., the first four bytes of the keccak256-hash of a function signature - in this case an error signature.

Control Structures: Solidity has *if*, *else*, *while*, *do*, *for*, *break*, *continue*, *return*, with the usual semantics known from C or JavaScript

- a. Parentheses can not be omitted for conditionals, but curly braces can be omitted around single-statement bodies
- b. Note that there is no type conversion from non-boolean to boolean types as there is in C and JavaScript, so *if (1) { ... }* is not valid Solidity.

- d. Solidity supports two error signatures: `Error(string)` and `Panic(uint256)`. The first (“error”) is used for “regular” error conditions while the second (“panic”) is used for errors that should not be present in bug-free code.

The low-level functions *call*, *delegatecall* and *staticcall* return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed.

The *assert* function creates an error of type `Panic(uint256)`. Assert should only be used to test for internal errors, and to check invariants. Properly functioning code should never create a `Panic`, not even on invalid external input.

A `Panic` exception is generated in the following situations. The error code supplied with the error data indicates the kind of panic:

- a. 0x01: If you call `assert` with an argument that evaluates to false.
- b. 0x11: If an arithmetic operation results in underflow or overflow outside of an unchecked `{ ... }` block.
- c. 0x12: If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
- d. 0x21: If you convert a value that is too big or negative into an enum type.
- e. 0x31: If you call `.pop()` on an empty array.

- f. 0x22: If you access a storage byte array that is incorrectly encoded.
- g. 0x32: If you access an array, `bytesN` or an array slice at an out-of-bounds or negative index (i.e. `x[i]` where `i >= x.length` or `i < 0`).
- h. 0x41: If you allocate too much memory or create an array that is too large.
- i. 0x51: If you call a zero-initialized variable of internal function type.

The *require* function either creates an error of type `Error(string)` or an error without any error data and it should be used to ensure valid conditions that cannot be detected until execution time. This includes conditions on inputs or return values from calls to external contracts. You can optionally provide a message string for *require*, but not for *assert*.

***revert*:** A direct revert can be triggered using the `revert` statement and the `revert` function. The `revert` statement takes a custom error as a direct argument without parentheses: `revert CustomError(arg1, arg2)`. The *revert()* function is another way to trigger exceptions from within other code blocks to flag an error and revert the current call. The function takes an optional string message containing details about the error that is passed back to the caller and it will create an `Error(string)` exception. Using a custom error instance will usually be much cheaper than a string description, because you can use the name of the error to describe it, which is encoded in only four bytes. A longer description can be supplied via `NatSpec` which does not incur any costs.

A ***Error(string)*** exception (or an exception without data) is generated in the following situations:

- Calling *require* with an argument that evaluates to false.
- If you perform an external function call targeting a contract that contains no code
- If your contract receives Ether via a public function without payable modifier (including the constructor and the fallback function)
- If your contract receives Ether via a public getter function

***try/catch*:** The *try* keyword has to be followed by an expression representing an external function call or a contract creation (*new ContractName()*). Errors inside the expression are not caught (for example if it is a complex expression that also involves internal function calls), only a revert happening inside the external call itself. The returns part (which is optional) that follows declares return variables matching the types returned by the external call. In case there was no error, these variables are assigned and the contract's execution continues inside the first success block. If the end of the success block is reached, execution continues after the *catch* blocks.

Solidity supports different kinds of catch blocks depending on the type of error:

- a. *catch Error(string memory reason) { ... }*: This catch clause is executed if the error was caused by `revert("reasonString")` or `require(false, "reasonString")` (or an internal error that causes such an exception).
- b. *catch { ... }*: If you are not interested in the error data, you can just use *catch { ... }* (even as the only catch clause) instead of the previous clause.

If execution reaches a catch-block, then the state-changing effects of the external call have been reverted. If execution reaches the success block, the effects were not reverted. If the effects have been reverted, then execution either continues in a catch block or the execution of the try/catch statement itself reverts (for example due to decoding failures as noted above or due to not providing a low-level catch clause).

- c. *catch Panic(uint errorCode) { ... }*: If the error was caused by a panic, i.e. by a failing assert, division by zero, invalid array access, arithmetic overflow and others, this catch clause will be run.
- d. *catch (bytes memory lowLevelData) { ... }*: This clause is executed if the error signature does not match any other clause, if there was an error while decoding the error message, or if no error data was provided with the exception. The declared variable provides access to the low-level error data in that case.

The reason behind a failed call can be manifold. Do not assume that the error message is coming directly from the called contract: The error might have happened deeper down in the call chain and the called contract just forwarded it. Also, it could be due to an out-of-gas situation and not a deliberate error condition: The caller always retains 63/64th of the gas in a call and thus even if the called contract goes out of gas, the caller still has some gas left

Solidity 10197	Solidity 10198 (1/2)
<p>Programming style: coding conventions for writing solidity code. Style is about consistency. Consistency with style is important. Consistency within a project is more important. Consistency within one module or function is most important. Two main categories: 1) Layout 2) Naming Conventions. Programming style affects readability and maintainability, both of which affect security.</p>	<p>Code Layout:</p> <ul style="list-style-type: none">a. Indentation: Use 4 spaces per indentation levelb. Tabs or Spaces: Spaces are the preferred indentation method. Mixing tabs and spaces should be avoided.c. Blank Lines: Surround top level declarations in solidity source with two blank lines.d. Maximum Line Length: Keeping lines to a maximum of 79 (or 99) characters helps readers easily parse the code.e. Wrapped lines should conform to the following guidelines: The first argument should not be attached to the opening parenthesis. One, and only one, indent should be used. Each argument should fall on its own line. The terminating element,),, should be placed on the final line by itself.
Solidity 10198 (2/2)	Solidity 10199 (1/2)
<ul style="list-style-type: none">f. Source File Encoding: UTF-8 or ASCII encoding is preferred.g. Imports: Import statements should always be placed at the top of the file.h. Order of Functions: Ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. Functions should be grouped according to their visibility and ordered: constructor, receive function (if exists), fallback function (if exists), external, public, internal, private. Within a grouping, place the view and pure functions last.	<p>More Code Layout:</p> <ul style="list-style-type: none">a. Whitespace in Expressions: Avoid extraneous whitespace in the following situations — Immediately inside parenthesis, brackets or braces, with the exception of single line function declarations.b. Control Structures: The braces denoting the body of a contract, library, functions and structs should: open on the same line as the declaration, close on their own line at the same indentation level as the beginning of the declaration. The opening brace should be preceded by a single space.c. Function Declaration: For short function declarations, it is recommended for the opening brace of the function body to be kept on the same line as the function declaration. The closing brace should be at the same indentation level as the function declaration. The opening brace should be preceded by a single space.

Solidity 101

99 (2/2)

- d.** Mappings: In variable declarations, do not separate the keyword mapping from its type by a space. Do not separate any nested mapping keyword from its type by whitespace.
- e.** Variable Declarations: Declarations of array variables should not have a space between the type and the brackets.
- f.** Strings should be quoted with double-quotes instead of single-quotes.
- g.** Operators: Surround operators with a single space on either side. Operators with a higher priority than others can exclude surrounding whitespace in order to denote precedence. This is meant to allow for improved readability for complex statements. You should always use the same amount of whitespace on either side of an operator
- h.** Layout contract elements in the following order: Pragma statements, Import statements, Interfaces, Libraries, Contracts. Inside each contract, library or interface, use the following order: Type declarations, State variables, Events, Functions

Solidity 101

100 (2/2)

- e.** Events should be named using the CapWords style. Examples: Deposit, Transfer, Approval, BeforeTransfer, AfterTransfer.
- f.** Functions should use mixedCase. Examples: getBalance, transfer, verifyOwner, addMember, changeOwner.

Solidity 101

100 (1/2)

Naming Convention:

- a.** Types: lowercase, lower_case_with_underscores, UPPERCASE, UPPER_CASE_WITH_UNDERSCORES, CapitalizedWords, mixedCase, Capitalized_Words_With_Underscores
- b.** Names to Avoid: l - Lowercase letter el, O - Uppercase letter oh, I - Uppercase letter eye. Never use any of these for single letter variable names. They are often indistinguishable from the numerals one and zero.
- c.** Contracts and libraries should be named using the CapWords style. Contract and library names should also match their filenames. If a contract file includes multiple contracts and/or libraries, then the filename should match the core contract. This is not recommended however if it can be avoided. Examples: SimpleToken, SmartBank, CertificateHashRepository, Player, Congress, Owned.
- d.** Structs should be named using the CapWords style. Examples: MyCoin, Position, PositionXY.