* Implicit vs Explicit — Definition
* Implicit vs Explicit Conversions in Solidity
* Conversion between unsigned integers uintN
* Conversion between bytesN (*eg: bytes4 <-> bytes16)*
* Conversions from uintM to bytesN
* Conversion from bytesN to uintM
* Conversion from bytes to string
* Table Summary — uintM ← → bytesN
* Literal assignments to bytesN
* Decimals and Hex Literals assignments uintN
* Conversion from bytes to bytesN
* Conversions to address type
* Conversion between address and address payable
* Conversion between Contract and address types
* References

**Implicit vs Explicit — Definition**

Before diving into *implicit vs explicit conversion* in Solidity, let’s first understand the difference in human language.

[Lexico.com](https://www.lexico.com/) gives the following definitions of implicit and explicit.

[Implicit](https://www.lexico.com/definition/implicit) = something is suggested, but not directly expressed.

[Explicit](https://www.lexico.com/definition/explicit) = something is stated clearly and in detail, leaving no room for confusion or doubt.

When something is explicit, it is very clear. There is no vague understanding or ambiguity

When something is implicit, it is implied. Something is understood from the wording, it is not directly stated or clearly described.

Let’s illustrate with an example. Alice is an employee of Bob. She is looking to express her decision to leave the company if she does not obtain a pay rise. She can express this either:

implicitly: *“if I am not being shown more appreciation, I will review my options.”*

explicitly: *“if I do not obtain a pay rise, I’ll leave.”*

**Implicit vs Explicit Conversions in Solidity**

All programming languages support some type of conversion.

Solidity also allows type conversion. Type conversion in Solidity can occur in three main scenarios:

* through variable assignments.
* when passing arguments to functions.
* when applying operators.

Type conversion can be done either implicitly (= the compiler derives the type automatically) or by being explicit to the compiler (= by telling the compiler which type to convert to).

Let’s look in detail at the underlying rules of implicit vs explicit conversions for the Solidity compiler.

[**Implicit Conversion**](https://docs.soliditylang.org/en/v0.8.13/types.html?highlight=Implicit%20conversion#implicit-conversions)

Implicit conversion between two data types is allowed in Solidity if:

* it makes sense semantically *(what does that mean?)*
* no information is lost in the process.

*Examples:*

* uint8 to uint16 = ✅
* int120 to int256 = ✅
* int8 to uint256 = ❌ (*because an uint256 cannot hold negative values that could potentially come from the int8).*

You can see from the last example above that the Solidity compiler does not allow implicit conversion when some information can potentially be lost.

To illustrate using the last example, if the value of the int8 was -5, the compiler would have to drop the negation in order to convert it to an allowed uint256 number. This conversion leads to information loss in the data, and the Solidity compiler is smart enough to know that and warn you.

However, if you *“do not agree with the compiler”*, or if you want to *“enforce some conversions”,* you can always tell the compiler what to do by being explicit.

**Explicit conversion**

*If the compiler does not allow implicit conversion but you know what you are doing, an explicit type conversion is sometimes possible.*

Explicit conversion can be done through *casting*or a constructor-like syntax.

uint256 a = 12345;  
bytes32 b = bytes32(a);

However, explicit conversion can be risky, as described in the Solidity docs.

*This may result in unexpected behaviour and allows you to bypass some security features of the compiler, so be sure to test that the result is what you want and expect!*

**Definitions summary**

Here is a two sentences summary of *implicit* *vs explicit* conversion in Solidity.

When converting from type A to type B in Solidity, some information in the data can be lost during the conversion process.

* with implicit conversion: you might not be aware of the (potential) information being lost.

If some information is actually lost, the compiler will refuse to compile and throws an error ❌

* with explicit conversion: you are fully aware that some information could be lost.

Since *you are being explicit to the compiler*, it will allow you to compile, and allows the (potential) loss of information ✅

**Conversion between unsigned integers**uintN

Unsigned integers in Solidity exist with different bits size, in sequences of 8 bits. Example: uint8, uint16, uint24, uint32, … up to uint256 .

For understanding, let’s define a **higher type**and **lower type** number:

* **larger type number:**a number closer to the lowest bits range uint8.
* **smaller type number:** a number closer to the highest bits range uint256.

The conversion for unsigned integers can go both ways.

**Converting to a larger type**

In this scenario, you are converting *a*number to a new type that is larger than the initial type.

*e.g:*uint64 to uint128

When converting an unsigned integer to a larger type,**left-padding occurs**, meaning zeroes (= 0 bits) are added to the left.

uint16 a = 0x1234;  
uint32 b = uint32(a); // b = 0x00001234

**Converting to a smaller type**

In this scenario, you are converting a number to a new type that is smaller than the initial type.

*e.g:*uint256 to uint128

When converting an unsigned integer to a smaller type, the **high order bits (the bits “the most on the left”) end up being discarded.**

*Example:*

uint32 a = 0x12345678;  
uint16 b = uint16(a); // b = 0x5678

This is the equivalent of doing a modulo of the number we want to convert with the higher number of the range of the new bits.

Let’s take the following examples:

uint32 a = 100000;  
uint16 public b = uint16(a); //b = a % 65536  
uint8 public c = uint8(a); //c = a % 256

In the above example:

* uint16 b can be calculated by doing a % 65536 = 34,464
* uint8 c can be calculated by doing a % 256 = 160

**Conversion between bytesN (*eg: bytes4 <-> bytes16)***

Solidity allows converting between different fixed-size bytes. Several scenarios exist. These are covered below.

**Converting to a smaller bytes range**

When explicitly converting to a smaller-bytes range, the right-most bytes are discarded (= the *“higher order bytes”).*

bytes2 a = 0x1234;  
bytes1 b = bytes1(a); // b = 0x12

This basically mean that Solidity truncates from the right hand side, until the length in bytes is equal to new length of the bytes specified in the type casting.

**Converting to a higher bytes range**

When converting to larger bytes, zero padding is added to the right.

bytes2 a = 0x1234;  
bytes4 b = bytes4(a); // b = 0x12340000

**Conversions from uintM to bytesN**

Below are the rules for converting two values a and b. Let’s use the following template to better understand the future possible conversions.

*uintM a , where M = a 8-bits range between uint8 ... uint256*

*bytesN b, where N = a 1-byte range between bytes1 ... bytes32*

**Implicit conversion**

uintM and bytesN cannot be implicitly converted between each other ❌

uint32 a = 0xcafecafe;  
bytes4 b = a; // TypeError// TypeError: Type uint32 is not implicitly convertible to expected type bytes4.  
bytes4 c = 0xbeefbeef;  
uint32 d = c; // TypeError// TypeError: Type bytes4 is not implicitly convertible to expected type uint32.

**Explicit conversion**

Explicit conversion is allowed as long as both theuintM and bytesN have the same size (the number of bits M is equivalent to the number of bytes N) ✅

e.g: M-bits = N-bytes

uint32 a = 0xcafecafe;  
bytes4 b = bytes4(a); // OKbytes4 c = 0xbeefbeef;  
uint32 d = uint32(c); // OK

e.g.: M-bits > N-bytes

uint32 a = 0xcafecafe;  
bytes3 b = bytes3(a); // TypeError// TypeError: Explicit type conversion not allowed from "uint32" to "bytes3".

e.g.: M-bits < N-bytes

uint32 a = 0xcafecafe;  
bytes5 b = bytes5(a); // TypeError// TypeError: Explicit type conversion not allowed from "uint32" to "bytes5".

**Conversion from bytesN to uintM**

Same rules as before but in the other order.

Explicit conversion is allowed as long as the bytesN and uintM are of the same size (number of bits M is equivalent to the number of bytes N)✅

e.g: N-bytes = M-bits

bytes4 a = 0xbeefbeef;  
uint32 b = uint32(a);

e.g: N-bytes > M-bits

bytes4 a = 0xbeefbeef;  
uint24 b = uint24(a); // TypeError// TypeError: Explicit type conversion not allowed from "bytes4" to "uint24".

e.g: N-bytes < M-bits

bytes4 a = 0xbeefbeef;  
uint40 b = uint40(a); // TypeError// TypeError: Explicit type conversion not allowed from "bytes4" to "uint40".

**Table summary — uintM ← → bytesN:**

The table below summarizes the equivalence between uintN and bytesN. Just remember that the number for uintM is the number of bits, the number for bytesN is the number of bytes, and one byte N = 8 bits M.

bytes ← → unit table equivalence for Solidity

**Hex Literals assignments to bytesN**

**Implicit assignment**

Any hexadecimal literal can be implicitly assigned to abytesN as long as the literal has the same number of bytes mentioned in the type.

bytes4 a = 0xcafecafe;bytes4 b = 0xcafe; // TypeError: Type [...] not implicitly convertible to expected type bytes4.bytes4 c = 0xcafecafecafe; // TypeError: Type [...] is not implicitly convertible to expected type bytes4.

**Decimals and Hex Literals assignments uintN**

**Implicit conversion**

Decimals or hexadecimal number literals can be implicitly converted to any uintN , but has to follow one of the following two rules:

* the uintN is the same size as the literal number ✅
* the uintN is of larger size than the literal number ✅

In summary, the rule is that integer type(= the range of the bits) has to be large enough to represent + hold value without truncation.

Here is the example from the Solidity docs:

uint8 a = 12; // no error  
uint32 b = 1234; // no error  
uint16 c = 0x123456; // error, as truncation required to 0x3456

**Explicit conversion**

Prior to Solidity 0.8.0. it was possible to explicitly convert any decimal or hexadecimal literal to any integer type (no matter the bits range). See the example below.

// this would compile up to solc 0.7.6  
uint8 a = uint8(300); // a = 44

The result of this explicit conversion would have been a equivalent to calculating the modulo of 300, as 300 % 256 = 44.

Since Solidity 0.8.0, the code above would result in the error:

TypeError: Explicit type conversion not allowed from "int\_const 300" to "uint8".

There such explicit conversions for literals are as strict as implicit conversions starting from 0.8.0. Meaning they are only allowed if the literal fits in the resulting range.

**Conversion from bytes to bytesN**

* Implicit conversion is not allowed ❌
* Explicit conversion is allowed since Solidity 0.8.5 🙌 🙂

Below is an example:

bytes memory data = new bytes(5);  
bytes2 firstTwoBytes = bytes2(data);

**Conversion between bytes and strings**

* Implicit conversion is not allowed ❌ on either side (bytes to string, or string to bytes)
* Explicit conversion is allowed ✅

string memory a = "All About Solidity";  
bytes memory b = bytes(a);bytes memory c = new bytes(5);  
string memory d = string(c);

Here is a practical example of a contract that uses this explicit type of conversion to convert from raw bytes to string: the [LSP4Compatibility.sol](https://github.com/lukso-network/lsp-smart-contracts/blob/b9f2315954a0ebc9b0ba2255ee6de33b93997f7c/contracts/LSP4DigitalAssetMetadata/LSP4Compatibility.sol) contract from [@lukso/lsp-smart-contracts](https://www.npmjs.com/package/@lukso/lsp-smart-contracts).

[LSP4 is a Metadata Standard](https://docs.lukso.tech/standards/nft-2.0/LSP4-Digital-Asset-Metadata) used to describe a token or NFT on

[LUKSO](https://medium.com/u/2376b006b57f?source=post_page-----661130eb8bec--------------------------------)

 (*see*[*LSP7*](https://github.com/lukso-network/LIPs/blob/main/LSPs/LSP-7-DigitalAsset.md)*or*[*LSP8*](https://github.com/lukso-network/LIPs/blob/main/LSPs/LSP-8-IdentifiableDigitalAsset.md)*for more details about this new generation of tokens and NFTs on EVM based chains*).

In LSP4, the basic information of a token or NFT (like its name or symbol) is stored under specific “data keys” in the [ERC725Y key-value](https://docs.lukso.tech/standards/universal-profile/lsp0-erc725account#erc725y---generic-key-value-store) store of the token /NFT.

These keys are mentioned in the code snippet below as \_LSP4\_TOKEN\_NAME\_KEY and \_LSP4\_TOKEN\_SYMBOL\_KEY.



source: [Github — lukso-network/lsp-smart-contracts, LSP4Compatibility.sol, lines 21–37](https://github.com/lukso-network/lsp-smart-contracts/blob/b9f2315954a0ebc9b0ba2255ee6de33b93997f7c/contracts/LSP4DigitalAssetMetadata/LSP4Compatibility.sol)

The code above is from the LSP4Compatibility.sol contract. This contract enables to create LSP7 tokens and LSP8 NFTs that are backwards compatible, meaning any ERC20 and ERC721 tokens can interact with them like they would with regular ERC20 / ERC721 tokens. (only difference is that [LSP7 and LSP8 have better built-in security + more extensible metadata!](https://docs.lukso.tech/standards/nft-2.0/introduction#how-tokens--nft-20-are-different) 😉)

Let’s go back to bytes to string conversion. In the code snippet above, the functions name() and symbol() retrieve data from the underlying ERC725Y key-value store, where all the data is stored as raw bytes.

To enable backwards compatibility, these bytes are explicitly converted to string (think of it like *casting).*This explicit conversion results in these two functions like the ones from the ERC20 / 721 standards, while the name and symbols are actually not stored under variables, but under the key-value storage abstraction obtained thanks to [ERC725Y](https://github.com/ERC725Alliance/ERC725/blob/main/docs/ERC-725.md#erc725y) 🗄

**Conversions to address type**

**Conversion from hex literals to address**

Below are the rules for converting a hexadecimal literal to an address, either *implicitly* via assignment or *explicitly* via *type casting*like address(0x…) .

Implicit conversion

Any hex literal can be implicitly converted to an address type if it passes the following two requirements:

rule 1: must have the correct size: 20 bytes long.

// not long enough  
address vanityAddress = 0xfccfdadf3acefddcdebdefad8d0e7cbb96eeee;// TypeError: Type int\_const 5637...(38 digits omitted)...7022 is not implicitly convertible to expected type address.

rule 2: must have a valid checksum

// invalid checksum  
address vanityAddress = 0xfccfdadf3acefddcdebdefad8d0e7cbb96eeeebf;// SyntaxError: This looks like an address but has an invalid checksum. Correct checksummed address: "0xFCCfDadf3acEFDdcdeBdefaD8d0e7Cbb96eeEeBf". If this is not used as an address, please prepend '00'. For more information please see https://docs.soliditylang.org/en/develop/types.html#address-literals

As you can see from above, the Solidity compiler will give you an error back but also the address literal back with a valid checksum.

The final code snippet will compile successfully, as it follows the 2 rules:

address vanityAddress = 0xFCCfDadf3acEFDdcdeBdefaD8d0e7Cbb96eeEeBf;

Explicit conversion

You can convert any hex literal explicitly to an address as shown below. This is allowed as long the literal is less or equal to 20 bytes.

If the hex literal is less than 20 bytes, it will left-zero pad the address + check-sum it.

address example1 = address(0xcafecafe)  
// 0x00000000000000000000000000000000CaFECAfEaddress example2 = address(0xca11ab1e00beef010101)  
// 0x00000000000000000000ca11AB1e00BEEF010101

If the hex literal is exactly 20 bytes long, the literal must have a valid checksum. Otherwise, the Solidity compiler will return an error and give you back the valid checksummed literal.

address example = address(0xcafecafecafecafecafecafecafecafecafecafe);// SyntaxError: This looks like an address but has an invalid checksum. Correct checksummed address: "0xCAfEcAfeCAfECaFeCaFecaFecaFECafECafeCaFe". If this is not used as an address, please prepend '00'. For more information please see <https://docs.soliditylang.org/en/develop/types.html#address-literals>

**Conversion from uint160 to address**

* Implicit conversion is not allowed from uint160 to address ❌
* Explicit conversion is allowed from uint160 to address ✅

uint160 someNumber = 5\_689\_454\_112;address convertedAddress = address(someNumber);

*NB: prior to Solidity 0.8.0 (up to Solidity 0.7.6), it was possible to convert explicitly any integer type uintN to an address (via casting). Since Solidity 0.8.0, explicit conversion is only allowed with uint160.*

**Conversion from address to uint160**

* Implicit conversion is not allowed from address to uint160 ❌
* Explicit conversion is allowed from address to uint160 ✅

This might seem an odd case, and not a very common one (*as far as I am aware, I have never seen such implementation).*The code below gives an illustrative example

function addressToUint160() public view returns (uint160) {  
   
 address from = msg.sender;  
 uint160 result = uint160(from);  
 return result;}// example with msg.sender = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4  
// result = 520786028573371803640530888255888666801131675076

**Conversion between address and address payable**

An address from can be explicitly converted to address payable via payable(from)

address payable from = msg.sender;// TypeError: Type address is not implicitly convertible to expected type address payable.  
address payable from = payable(msg.sender);

This is especially relevant to our previous examples, as any explicit conversion into address type (using address(…)) always returns a non-payable address type.

Therefore, any Address Literal, bytes20 or uint160 value can be explicitly converted to an address payable as follow:

// conversion from Address Literal to address payableaddress to = 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4;  
address payable payableTo = payable(to);// conversion from bytes20 to address payablebytes20 from = bytes20(0x5B38Da6a701c568545dCfcB03FcB875f56beddC4);  
address payable payableFrom = payable(address(from));  
// conversion from uint160 to address payableuint160 u = 12345;  
address payable converted = payable(address(u));

**Conversion between Contract and address types**

Explicit conversion can be done from an address to a Contract type.

Let’s take the example below from the Solidity docs:

address creator = TokenCreator(msg.sender);

Through this explicit conversion, we assume here that the type of msg sender (= the calling contract) is TokenCreator. However, there is no real way to verify this apart from using ERC165 Standard Interface detection (assuming that msg.sender here implements this standard).