Expressions and Control Structures

Control Structures

Most of the control structures known from curly-braces languages are available in Solidity:

There is: `if`, `else`, `while`, `do`, `for`, `break`, `continue`, `return`, with the usual semantics known from C or JavaScript.

Solidity also supports exception handling in the form of `try`/`catch`-statements, but only for external function calls and contract creation calls. Errors can be created using the revert statement.

Parentheses can *not* be omitted for conditionals, but curly braces can be omitted around single-statement bodies.

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.

Function Calls

Internal Function Calls

Functions of the current contract can be called directly ("internally"), also recursively, as seen in this nonsensical example:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

// This will report a warning
contract C {
    function g(uint a) public pure returns (uint ret) { return a + f(); }
    function f() internal pure returns (uint ret) { return g(7) + f(); }
}
```

These function calls are translated into simple jumps inside the EVM. This has the effect that the current memory is not cleared, i.e. passing memory references to internally-called functions is very efficient. Only functions of the same contract instance can be called internally.

You should still avoid excessive recursion, as every internal function call uses up at least one stack slot and there are only 1024 slots available.

External Function Calls

Functions can also be called using the `this.g(8);` and `c.g(2);` notation, where `c` is a contract instance and `g` is a function belonging to `c`. Calling the function `g` via either way results in it being called "externally", using a message call and not directly via jumps. Please note that function calls on `this` cannot be used in the constructor, as the actual contract has not been created yet.

Functions of other contracts have to be called externally. For an external call, all function arguments have to be copied to memory.

> **Note**
>
> A function call from one contract to another does not create its own transaction, it is a message call as part of the overall transaction.

When calling functions of other contracts, you can specify the amount of Wei or gas sent with the call with the special options `{value: 10, gas: 10000}`. Note that it is discouraged to specify gas values explicitly, since the gas costs of opcodes can change in the future. Any Wei you send to the contract is added to the total balance of that contract:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.6.2 <0.9.0;

contract InfoFeed {
    function info() public payable returns (uint ret) { return 42; }
}

contract Consumer {
    InfoFeed feed;
    function setFeed(InfoFeed addr) public { feed = addr; }
    function callFeed() public { feed.info{value: 10, gas: 800}(); }
}
```

You need to use the modifier `payable` with the `info` function because otherwise, the `value` option would not be available.

Warning

Be careful that `feed.info{value: 10, gas: 800}` only locally sets the `value` and amount of `gas` sent with the function call, and the parentheses at the end perform the actual call. So `feed.info{value: 10, gas: 800}` does not call the function and the `value` and `gas` settings are lost, only `feed.info{value: 10, gas: 800}()` performs the function call.

Due to the fact that the EVM considers a call to a non-existing contract to always succeed, Solidity uses the `extcodesize` opcode to check that the contract that is about to be called actually exists (it contains code) and causes an exception if it does not. This check is skipped if the return data will be decoded after the call and thus the ABI decoder will catch the case of a non-existing contract.

Note that this check is not performed in case of low-level calls which operate on addresses rather than contract instances.

Note

Be careful when using high-level calls to precompiled contracts, since the compiler considers them non-existing according to the above logic even though they execute code and can return data.

Function calls also cause exceptions if the called contract itself throws an exception or goes out of gas.

Warning

Any interaction with another contract imposes a potential danger, especially if the source code of the contract is not known in advance. The current contract hands over control to the called contract and that may potentially do just about anything. Even if the called contract inherits from a known parent contract, the inheriting contract is only required to have a correct interface. The implementation of the contract, however, can be completely arbitrary and thus, pose a danger. In addition, be prepared in case it calls into other contracts of your system or even back into the calling contract before the first call returns. This means that the called contract can change state variables of the calling contract via its functions. Write your functions in a way that, for example, calls to external functions happen after any changes to state variables in your contract so your contract is not vulnerable to a reentrancy exploit.

Note

Before Solidity 0.6.2, the recommended way to specify the value and gas was to use `f.value(x).gas(g)()`. This was deprecated in Solidity 0.6.2 and is no longer possible since Solidity 0.7.0.

Named Calls and Anonymous Function Parameters

Function call arguments can be given by name, in any order, if they are enclosed in `{ }` as can be seen in the following example. The argument list has to coincide by name with the list of parameters from the function declaration, but can be in arbitrary order.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.0 <0.9.0;

contract C {
    mapping(uint => uint) data;

    function f() public {
        set({value: 2, key: 3});
    }

    function set(uint key, uint value) public {
        data[key] = value;
    }

}
```

Omitted Function Parameter Names

The names of unused parameters (especially return parameters) can be omitted. Those parameters will still be present on the stack, but they are inaccessible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    // omitted name for parameter
    function func(uint k, uint) public pure returns(uint) {
        return k;
    }
}
```

Creating Contracts via `new`

A contract can create other contracts using the `new` keyword. The full code of the contract being created has to be known when the creating contract is compiled so recursive creation-dependencies are not possible.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) payable {
        x = a;
    }
}

contract C {
    D d = new D(4); // will be executed as part of C's constructor

    function createD(uint arg) public {
        D newD = new D(arg);
        newD.x();
    }

    function createAndEndowD(uint arg, uint amount) public payable {
        // Send ether along with the creation
        D newD = new D{value: amount}(arg);
        newD.x();
    }
}
```

As seen in the example, it is possible to send Ether while creating an instance of `D` using the `value` option, but it is not possible to limit the amount of gas. If the creation fails (due to out-of-stack, not enough balance or other problems), an exception is thrown.

Salted contract creations / create2

When creating a contract, the address of the contract is computed from the address of the creating contract and a counter that is increased with each contract creation.

If you specify the option `salt` (a bytes32 value), then contract creation will use a different mechanism to come up with the address of the new contract:

It will compute the address from the address of the creating contract, the given salt value, the (creation) bytecode of the created contract and the constructor arguments.

In particular, the counter ("nonce") is not used. This allows for more flexibility in creating contracts: You are able to derive the address of the new contract before it is created. Furthermore, you can rely on this address also in case the creating contracts creates other contracts in the meantime.

The main use-case here is contracts that act as judges for off-chain interactions, which only need to be created if there is a dispute.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;
contract D {
    uint public x;
    constructor(uint a) {
        x = a;
    }
}

contract C {
    function createDSalted(bytes32 salt, uint arg) public {
        // This complicated expression just tells you how the address
        // can be pre-computed. It is just there for illustration.
        // You actually only need ``new D{salt: salt}(arg)``.
        address predictedAddress = address(uint160(uint(keccak256(abi.encodePacked(
            bytes1(0xff),
            address(this),
            salt,
            keccak256(abi.encodePacked(
                type(D).creationCode,
                arg
            ))
        )))));

        D d = new D{salt: salt}(arg);
        require(address(d) == predictedAddress);
    }
}
```

Warning

There are some peculiarities in relation to salted creation. A contract can be re-created at the same address after having been destroyed. Yet, it is possible for that newly created contract to have a different deployed bytecode even though the creation bytecode has been the same (which is a requirement because otherwise the address would change). This is due to the fact that the constructor can query external state that might have changed between the two creations and incorporate that into the deployed bytecode before it is stored.

Order of Evaluation of Expressions

The evaluation order of expressions is not specified (more formally, the order in which the children of one node in the expression tree are evaluated is not specified, but they are of course evaluated before the node itself). It is only guaranteed that statements are executed in order and short-circuiting for boolean expressions is done.

Assignment

Destructuring Assignments and Returning Multiple Values

Solidity internally allows tuple types, i.e. a list of objects of potentially different types whose number is a constant at compile-time. Those tuples can be used to return multiple values at the same time. These can then either be assigned to newly declared variables or to pre-existing variables (or LValues in general).

Tuples are not proper types in Solidity, they can only be used to form syntactic groupings of expressions.

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

contract C {
    uint index;

    function f() public pure returns (uint, bool, uint) {
        return (7, true, 2);
    }

    function g() public {
        // Variables declared with type and assigned from the returned tuple,
        // not all elements have to be specified (but the number must match).
        (uint x, , uint y) = f();
        // Common trick to swap values -- does not work for non-value storage types.
        (x, y) = (y, x);
        // Components can be left out (also for variable declarations).
        (index, , ) = f(); // Sets the index to 7
    }
}
```

It is not possible to mix variable declarations and non-declaration assignments, i.e. the following is not valid: `(x, uint y) = (1, 2);`

Note

Prior to version 0.5.0 it was possible to assign to tuples of smaller size, either filling up on the left or on the right side (which ever was empty). This is now disallowed, so both sides have to have the same number of components.

Warning

Be careful when assigning to multiple variables at the same time when reference types are involved, because it could lead to unexpected copying behaviour.

Complications for Arrays and Structs

The semantics of assignments are more complicated for non-value types like arrays and structs, including `bytes` and `string`, see Data location and assignment behaviour for details.

In the example below the call to `g(x)` has no effect on `x` because it creates an independent copy of the storage value in memory. However, `h(x)` successfully modifies `x` because only a reference and not a copy is passed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.4.22 <0.9.0;

contract C {
    uint[20] x;

    function f() public {
        g(x);
        h(x);
    }

    function g(uint[20] memory y) internal pure {
        y[2] = 3;
    }

    function h(uint[20] storage y) internal {
        y[3] = 4;
    }
}
```

Scoping and Declarations

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.

Scoping in Solidity follows the widespread scoping rules of C99 (and many other languages): 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.

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.

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.

As a consequence, the following examples will compile without warnings, since the two variables have the same name but disjoint scopes.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
contract C {
    function minimalScoping() pure public {
        {
            uint same;
            same = 1;
        }

        {
            uint same;
            same = 3;
        }
    }
}
```

As a special example of the C99 scoping rules, note that in the following, the first assignment to `x` will actually assign the outer and not the inner variable. In any case, you will get a warning about the outer variable being shadowed.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will report a warning
contract C {
    function f() pure public returns (uint) {
        uint x = 1;
        {
            x = 2; // this will assign to the outer variable
            uint x;
        }
        return x; // x has value 2
    }
}
```

Warning

Before version 0.5.0 Solidity followed the same scoping rules as JavaScript, that is, a variable declared anywhere within a function would be in scope for the entire function, regardless where it was declared. The following example shows a code snippet that used to compile but leads to an error starting from version 0.5.0.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.5.0 <0.9.0;
// This will not compile
contract C {
    function f() pure public returns (uint) {
        x = 2;
        uint x;
        return x;
    }
}
```

Checked or Unchecked Arithmetic

An overflow or underflow is the situation where the resulting value of an arithmetic operation, when executed on an unrestricted integer, falls outside the range of the result type.

Prior to Solidity 0.8.0, arithmetic operations would always wrap in case of under- or overflow leading to widespread use of libraries that introduce additional checks.

Since Solidity 0.8.0, all arithmetic operations revert on over- and underflow by default, thus making the use of these libraries unnecessary.

To obtain the previous behaviour, an `unchecked` block can be used:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.0;
contract C {
    function f(uint a, uint b) pure public returns (uint) {
        // This subtraction will wrap on underflow.
        unchecked { return a - b; }
    }
    function g(uint a, uint b) pure public returns (uint) {
        // This subtraction will revert on underflow.
        return a - b;
    }
}
```

The call to `f(2, 3)` will return `2**256-1`, while `g(2, 3)` will cause a failing assertion.

The `unchecked` block can be used everywhere inside a block, but not as a replacement for a block. It also cannot be nested.

The setting only affects the statements that are syntactically inside the block. Functions called from within an `unchecked` block do not inherit the property.

Note

To avoid ambiguity, you cannot use `_;` inside an `unchecked` block.

The following operators will cause a failing assertion on overflow or underflow and will wrap without an error if used inside an unchecked block:

`++`, `--`, `+`, binary `-`, unary `-`, `*`, `/`, `%`, `**`

`+=`, `-=`, `*=`, `/=`, `%=`

Warning

It is not possible to disable the check for division by zero or modulo by zero using the `unchecked` block.

Note

Bitwise operators do not perform overflow or underflow checks. This is particularly visible when using bitwise shifts (`<<` , `>>` , `<<=` , `>>=`) in place of integer division and multiplication by a power of 2. For example `type(uint256).max << 3` does not revert even though `type(uint256).max * 8` would.

Note

The second statement in `int x = type(int).min; -x;` will result in an overflow because the negative range can hold one more value than the positive range.

Explicit type conversions will always truncate and never cause a failing assertion with the exception of a conversion from an integer to an enum type.

Error handling: Assert, Require, Revert and 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.

When exceptions happen in a sub-call, they "bubble up" (i.e., exceptions are rethrown) automatically unless they are caught in a `try/catch` statement. 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".

Warning

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.

Exceptions can contain error data that is passed back to the caller in the form of error instances. The built-in errors `Error(string)` and `Panic(uint256)` are used by special functions, as explained below. `Error` is used for "regular" error conditions while `Panic` is used for errors that should not be present in bug-free code.

Panic via `assert` and Error via `require`

The convenience functions `assert` and `require` can be used to check for conditions and throw an exception if the condition is not met.

The `assert` function creates an error of type `Panic(uint256)` . The same error is created by the compiler in certain situations as listed below.

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. If this happens, then there is a bug in your contract which you should fix. Language analysis tools can evaluate your contract to identify the conditions and function calls which will cause a Panic.

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

1. 0x00: Used for generic compiler inserted panics.
2. 0x01: If you call `assert` with an argument that evaluates to false.
3. 0x11: If an arithmetic operation results in underflow or overflow outside of an `unchecked { ... }` block.
4. 0x12; If you divide or modulo by zero (e.g. `5 / 0` or `23 % 0`).
5. 0x21: If you convert a value that is too big or negative into an enum type.
6. 0x22: If you access a storage byte array that is incorrectly encoded.
7. 0x31: If you call `.pop()` on an empty array.
8. 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`).
9. 0x41: If you allocate too much memory or create an array that is too large.

10. 0x51: If you call a zero-initialized variable of internal function type.

The `require` function either creates an error without any data or an error of type `Error(string)`. 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.

> **Note**
>
> It is currently not possible to use custom errors in combination with `require`. Please use `if (!condition) revert CustomError();` instead.

An `Error(string)` exception (or an exception without data) is generated by the compiler in the following situations:

1. Calling `require(x)` where `x` evaluates to `false`.
2. If you use `revert()` or `revert("description")`.
3. If you perform an external function call targeting a contract that contains no code.
4. If your contract receives Ether via a public function without `payable` modifier (including the constructor and the fallback function).
5. If your contract receives Ether via a public getter function.

For the following cases, the error data from the external call (if provided) is forwarded. This means that it can either cause an *Error* or a *Panic* (or whatever else was given):

1. If a `.transfer()` fails.
2. If you call a function via a message call but it does not finish properly (i.e., it runs out of gas, has no matching function, or throws an exception itself), except when a low level operation `call`, `send`, `delegatecall`, `callcode` or `staticcall` is used. The low level operations never throw exceptions but indicate failures by returning `false`.
3. If you create a contract using the `new` keyword but the contract creation does not finish properly.

You can optionally provide a message string for `require`, but not for `assert`.

> **Note**
>
> If you do not provide a string argument to `require`, it will revert with empty error data, not even including the error selector.

The following example shows how you can use `require` to check conditions on inputs and `assert` for internal error checking.

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

contract Sharer {
    function sendHalf(address payable addr) public payable returns (uint balance) {
        require(msg.value % 2 == 0, "Even value required.");
        uint balanceBeforeTransfer = address(this).balance;
        addr.transfer(msg.value / 2);
        // Since transfer throws an exception on failure and
        // cannot call back here, there should be no way for us to
        // still have half of the money.
        assert(address(this).balance == balanceBeforeTransfer - msg.value / 2);
        return address(this).balance;
    }
}
```

Internally, Solidity performs a revert operation (instruction `0xfd`). This causes the EVM to revert all changes made to the state. The reason for reverting is that there is no safe way to continue execution, because an expected effect did not occur. Because we want to keep the atomicity of transactions, the safest action is to revert all changes and make the whole transaction (or at least call) without effect.

In both cases, the caller can react on such failures using `try` / `catch`, but the changes in the caller will always be reverted.

> Note
>
> Panic exceptions used to use the `invalid` opcode before Solidity 0.8.0, which consumed all gas available to the call. Exceptions that use `require` used to consume all gas until before the Metropolis release.

`revert`

A direct revert can be triggered using the `revert` statement and the `revert` function.

The `revert` statement takes a custom error as direct argument without parentheses:

```
revert CustomError(arg1, arg2);
```

For backwards-compatibility reasons, there is also the `revert()` function, which uses parentheses and accepts a string:

```
revert(); revert("description");
```

The error data will be passed back to the caller and can be caught there. Using `revert()` causes a revert without any error data while `revert("description")` will create an `Error(string)` error.

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.

The following example shows how to use an error string and a custom error instance together with `revert` and the equivalent `require`:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.8.4;

contract VendingMachine {
    address owner;
    error Unauthorized();
    function buy(uint amount) public payable {
        if (amount > msg.value / 2 ether)
            revert("Not enough Ether provided.");
        // Alternative way to do it:
        require(
            amount <= msg.value / 2 ether,
            "Not enough Ether provided."
        );
        // Perform the purchase.
    }
    function withdraw() public {
        if (msg.sender != owner)
            revert Unauthorized();

        payable(msg.sender).transfer(address(this).balance);
    }
}
```

The two ways `if (!condition) revert(...);` and `require(condition, ...);` are equivalent as long as the arguments to `revert` and `require` do not have side-effects, for example if they are just strings.

> Note
>
> The `require` function is evaluated just as any other function. This means that all arguments are evaluated before the function itself is executed. In particular, in `require(condition, f())` the function `f` is executed even if `condition` is true.

The provided string is abi-encoded as if it were a call to a function `Error(string)`. In the above example, `revert("Not enough Ether provided.");` returns the following hexadecimal as error return data:

```
0x08c379a0                                                         // Function selector for
Error(string)
0x0000000000000000000000000000000000000000000000000000000000000020 // Data offset
0x000000000000000000000000000000000000000000000000000000000000001a // String length
0x4e6f7420656e6f7567682045746865722070726f76696465642e000000000000 // String data
```

The provided message can be retrieved by the caller using `try` / `catch` as shown below.

> Note
>
> There used to be a keyword called `throw` with the same semantics as `revert()` which was deprecated in version 0.4.13 and removed in version 0.5.0.

`try` / `catch`

A failure in an external call can be caught using a try/catch statement, as follows:

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.8.1;

interface DataFeed { function getData(address token) external returns (uint value); }

contract FeedConsumer {
    DataFeed feed;
    uint errorCount;
    function rate(address token) public returns (uint value, bool success) {
        // Permanently disable the mechanism if there are
        // more than 10 errors.
        require(errorCount < 10);
        try feed.getData(token) returns (uint v) {
            return (v, true);
        } catch Error(string memory /*reason*/) {
            // This is executed in case
            // revert was called inside getData
            // and a reason string was provided.
            errorCount++;
            return (0, false);
        } catch Panic(uint /*errorCode*/) {
            // This is executed in case of a panic,
            // i.e. a serious error like division by zero
            // or overflow. The error code can be used
            // to determine the kind of error.
            errorCount++;
            return (0, false);
        } catch (bytes memory /*lowLevelData*/) {
            // This is executed in case revert() was used.
            errorCount++;
            return (0, false);
        }
    }
}
```

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:

- `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).

- `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.
- `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.
- `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.

It is planned to support other types of error data in the future. The strings `Error` and `Panic` are currently parsed as is and are not treated as identifiers.

In order to catch all error cases, you have to have at least the clause `catch { ...}` or the clause `catch (bytes memory lowLevelData) { ... }`.

The variables declared in the `returns` and the `catch` clause are only in scope in the block that follows.

Note

If an error happens during the decoding of the return data inside a try/catch-statement, this causes an exception in the currently executing contract and because of that, it is not caught in the catch clause. If there is an error during decoding of `catch Error(string memory reason)` and there is a low-level catch clause, this error is caught there.

Note

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).

Note

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.