Lecture 20 - Smart Contracts - Standard Contracts (ERC-20)

Standard ERC-20 Contract Interface

SimpleToken ERC20 Interface

Method Name	Const	$	Params
Approval	event		`( address owner, address spender, uint256 value )`
INITIAL_SUPPLY	const		`() returns ( uint256 )`
Transfer	event		`( address from, address to, uint256 value )`
allowance	const		`( address _owner, address _spender ) returns ( uint256 )`
approve	Tx		`( address _spender, uint256 _value ) returns ( bool )`
balanceOf	const		`( address _owner ) returns ( uint256 )`
decimals	const		`() returns ( uint8 )`
decreaseApproval	Tx		`( address _spender, uint256 _subtractedValue ) returns ( bool )`
increaseApproval	Tx		`( address _spender, uint256 _addedValue ) returns ( bool )`
name	const		`() returns ( string )`
symbol	const		`() returns ( string )`
totalSupply	const		`() returns ( uint256 )`
transfer	Tx		`( address _to, uint256 _value ) returns ( bool )`
transferFrom	Tx		`( address _from, address _to, uint256 _value ) returns ( bool )`
constructor	()		

Simple777Token Ours derived from ERC777

The ERC777 is a modern upgrade (and backward compatible) version of the ERC20 fungible token contract. It improves on ERC20 with a pair of hooks, tokensToSend and tokensReceived that address a number of concurrency problems in token transfer. It maintains compatibility with the above set of interfaces for the ERC20 contract.

Let's take a look at the derived contract.

File: ./eth/contracts/Simple777Token.sol

```
 1: // SPDX-License-Identifier: MIT
 2: pragma solidity >=0.4.22 <0.9.0;
 3:
 4: import "@openzeppelin/contracts/token/ERC777/ERC777.sol";
 5:
 6: /**
 7:  * @title Simple777Token
 8:  * @dev Very simple ERC777 Token example, where all tokens are pre-assigned to the creator.
 9:  * Note they can later distribute these tokens as they wish using `transfer` and other
10:  * `ERC20` or `ERC777` functions.
11:  * Based on https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/examples/SimpleToken.sol
12:  */
13: contract Simple777Token is ERC777 {
14:
15:     /**
16:      * @dev Constructor that gives msg.sender all of existing tokens.
17:      */
18:     constructor () ERC777("Simple777Token", "S7", new address[](0)) {
19:         _mint(msg.sender, 10000 * 10 ** 18, "", "", true);
20:     }
21: }
```

We are going to use the solc 8.1 compiler. To do this we need

1. A current install of openzeppelin

```
$ npm install @openzeppelin/contracts
```

2. Force truffle to use the 8.1 version of the solc compiler.

```
// Configure your compilers
compilers: {
solc: {
  optimizer: {
    enabled: true,
    runs: 200
  },
  version: "0.8.1",    // Fetch exact version from solc-bin (default: truffle's version)
  evmVersion: "petersburg"
}
},
```

In truffle-config.js

The Interface ERC777Token

```
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts v4.4.1 (token/ERC777/IERC777.sol)

pragma solidity ^0.8.0;

/**
 * @dev Interface of the ERC777Token standard as defined in the EIP.
 *
 * This contract uses the
 * https://eips.ethereum.org/EIPS/eip-1820[ERC1820 registry standard] to let
 * token holders and recipients react to token movements by using setting implementers
 * for the associated interfaces in said registry. See {IERC1820Registry} and
 * {ERC1820Implementer}.
 */
interface IERC777 {
    /**
     * @dev Returns the name of the token.
     */
    function name() external view returns (string memory);

    /**
     * @dev Returns the symbol of the token, usually a shorter version of the
     * name.
     */
    function symbol() external view returns (string memory);

    /**
     * @dev Returns the smallest part of the token that is not divisible. This
     * means all token operations (creation, movement and destruction) must have
     * amounts that are a multiple of this number.
     *
     * For most token contracts, this value will equal 1.
     */
    function granularity() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens in existence.
     */
    function totalSupply() external view returns (uint256);

    /**
     * @dev Returns the amount of tokens owned by an account (`owner`).
     */
    function balanceOf(address owner) external view returns (uint256);

    /**
     * @dev Moves `amount` tokens from the caller's account to `recipient`.
     *
     * If send or receive hooks are registered for the caller and `recipient`,
     * the corresponding functions will be called with `data` and empty
     * `operatorData`. See {IERC777Sender} and {IERC777Recipient}.
     *
     * Emits a {Sent} event.
     *
     * Requirements
     *
     * - the caller must have at least `amount` tokens.
     * - `recipient` cannot be the zero address.
     * - if `recipient` is a contract, it must implement the {IERC777Recipient}
     * interface.
     */
    function send(
        address recipient,
        uint256 amount,
        bytes calldata data
    ) external;

    /**
```

```
68:     /**
69:      * @dev Destroys `amount` tokens from the caller's account, reducing the
70:      * total supply.
71:      *
72:      * If a send hook is registered for the caller, the corresponding function
73:      * will be called with `data` and empty `operatorData`. See {IERC777Sender}.
74:      *
75:      * Emits a {Burned} event.
76:      *
77:      * Requirements
78:      *
79:      * - the caller must have at least `amount` tokens.
80:      */
81:     function burn(uint256 amount, bytes calldata data) external;
82:
83:     /**
84:      * @dev Returns true if an account is an operator of `tokenHolder`.
85:      * Operators can send and burn tokens on behalf of their owners. All
86:      * accounts are their own operator.
87:      *
88:      * See {operatorSend} and {operatorBurn}.
89:      */
90:     function isOperatorFor(address operator, address tokenHolder) external view returns (bool);
91:
92:     /**
93:      * @dev Make an account an operator of the caller.
94:      *
95:      * See {isOperatorFor}.
96:      *
97:      * Emits an {AuthorizedOperator} event.
98:      *
99:      * Requirements
100:      *
101:      * - `operator` cannot be calling address.
102:      */
103:     function authorizeOperator(address operator) external;
104:
105:     /**
106:      * @dev Revoke an account's operator status for the caller.
107:      *
108:      * See {isOperatorFor} and {defaultOperators}.
109:      *
110:      * Emits a {RevokedOperator} event.
111:      *
112:      * Requirements
113:      *
114:      * - `operator` cannot be calling address.
115:      */
116:     function revokeOperator(address operator) external;
117:
118:     /**
119:      * @dev Returns the list of default operators. These accounts are operators
120:      * for all token holders, even if {authorizeOperator} was never called on
121:      * them.
122:      *
123:      * This list is immutable, but individual holders may revoke these via
124:      * {revokeOperator}, in which case {isOperatorFor} will return false.
125:      */
126:     function defaultOperators() external view returns (address[] memory);
127:
128:     /**
129:      * @dev Moves `amount` tokens from `sender` to `recipient`. The caller must
130:      * be an operator of `sender`.
131:      *
132:      * If send or receive hooks are registered for `sender` and `recipient`,
133:      * the corresponding functions will be called with `data` and
134:      * `operatorData`. See {IERC777Sender} and {IERC777Recipient}.
135:      *
136:      * Emits a {Sent} event.
137:      *
138:      * Requirements
139:      *
140:      * - `sender` cannot be the zero address.
```

```
140:      * - `sender` cannot be the zero address.
141:      * - `sender` must have at least `amount` tokens.
142:      * - the caller must be an operator for `sender`.
143:      * - `recipient` cannot be the zero address.
144:      * - if `recipient` is a contract, it must implement the {IERC777Recipient}
145:      * interface.
146:      */
147:     function operatorSend(
148:         address sender,
149:         address recipient,
150:         uint256 amount,
151:         bytes calldata data,
152:         bytes calldata operatorData
153:     ) external;
154:
155:     /**
156:      * @dev Destroys `amount` tokens from `account`, reducing the total supply.
157:      * The caller must be an operator of `account`.
158:      *
159:      * If a send hook is registered for `account`, the corresponding function
160:      * will be called with `data` and `operatorData`. See {IERC777Sender}.
161:      *
162:      * Emits a {Burned} event.
163:      *
164:      * Requirements
165:      *
166:      * - `account` cannot be the zero address.
167:      * - `account` must have at least `amount` tokens.
168:      * - the caller must be an operator for `account`.
169:      */
170:     function operatorBurn(
171:         address account,
172:         uint256 amount,
173:         bytes calldata data,
174:         bytes calldata operatorData
175:     ) external;
176:
177:     event Sent(
178:         address indexed operator,
179:         address indexed from,
180:         address indexed to,
181:         uint256 amount,
182:         bytes data,
183:         bytes operatorData
184:     );
185:
186:     event Minted(address indexed operator, address indexed to, uint256 amount, bytes data, bytes operatorData);
187:
188:     event Burned(address indexed operator, address indexed from, uint256 amount, bytes data, bytes operatorData);
189:
190:     event AuthorizedOperator(address indexed operator, address indexed tokenHolder);
191:
192:     event RevokedOperator(address indexed operator, address indexed tokenHolder);
193: }
```

The biggest (and most important) difference between ERC20 and ERC777 is the addition of *operators*. The contract owner (operator) can authorize and revoke trusted entities - with the power to act on the owners behalf. The authorized entity can move tokens for all addresses.

`send` replaces `transfer` and `transferFrom`.

The contract...

```
 1: // SPDX-License-Identifier: MIT
 2: // OpenZeppelin Contracts (last updated v4.5.0) (token/ERC777/ERC777.sol)
 3:
 4: pragma solidity ^0.8.0;
 5:
 6: import "./IERC777.sol";
 7: import "./IERC777Recipient.sol";
 8: import "./IERC777Sender.sol";
 9: import "../ERC20/IERC20.sol";
10: import "../../utils/Address.sol";
11: import "../../utils/Context.sol";
12: import "../../utils/introspection/IERC1820Registry.sol";
13:
14: /**
15:  * @dev Implementation of the {IERC777} interface.
16:  *
17:  * This implementation is agnostic to the way tokens are created. This means
18:  * that a supply mechanism has to be added in a derived contract using {_mint}.
19:  *
20:  * Support for ERC20 is included in this contract, as specified by the EIP: both
21:  * the ERC777 and ERC20 interfaces can be safely used when interacting with it.
22:  * Both {IERC777-Sent} and {IERC20-Transfer} events are emitted on token
23:  * movements.
24:  *
25:  * Additionally, the {IERC777-granularity} value is hard-coded to `1`, meaning that there
26:  * are no special restrictions in the amount of tokens that created, moved, or
27:  * destroyed. This makes integration with ERC20 applications seamless.
28:  */
29: contract ERC777 is Context, IERC777, IERC20 {
30:     using Address for address;
31:
32:     IERC1820Registry internal constant _ERC1820_REGISTRY = IERC1820Registry(0x1820a4B7618BdE71Dce8cdc73aAB6C95905faD24);
33:
34:     mapping(address => uint256) private _balances;
35:
36:     uint256 private _totalSupply;
37:
38:     string private _name;
39:     string private _symbol;
40:
41:     bytes32 private constant _TOKENS_SENDER_INTERFACE_HASH = keccak256("ERC777TokensSender");
42:     bytes32 private constant _TOKENS_RECIPIENT_INTERFACE_HASH = keccak256("ERC777TokensRecipient");
43:
44:     // This isn't ever read from - it's only used to respond to the defaultOperators query.
45:     address[] private _defaultOperatorsArray;
46:
47:     // Immutable, but accounts may revoke them (tracked in __revokedDefaultOperators).
48:     mapping(address => bool) private _defaultOperators;
49:
50:     // For each account, a mapping of its operators and revoked default operators.
51:     mapping(address => mapping(address => bool)) private _operators;
52:     mapping(address => mapping(address => bool)) private _revokedDefaultOperators;
53:
54:     // ERC20-allowances
55:     mapping(address => mapping(address => uint256)) private _allowances;
56:
57:     /**
58:      * @dev `defaultOperators` may be an empty array.
59:      */
60:     constructor(
61:         string memory name_,
62:         string memory symbol_,
63:         address[] memory defaultOperators_
64:     ) {
65:         _name = name_;
66:         _symbol = symbol_;
67:
68:         _defaultOperatorsArray = defaultOperators_;
```

```
69:         for (uint256 i = 0; i < defaultOperators_.length; i++) {
70:             _defaultOperators[defaultOperators_[i]] = true;
71:         }
72:
73:         // register interfaces
74:         _ERC1820_REGISTRY.setInterfaceImplementer(address(this), keccak256("ERC777Token"), address(this));
75:         _ERC1820_REGISTRY.setInterfaceImplementer(address(this), keccak256("ERC20Token"), address(this));
76:     }
77:
78:     /**
79:      * @dev See {IERC777-name}.
80:      */
81:     function name() public view virtual override returns (string memory) {
82:         return _name;
83:     }
84:
85:     /**
86:      * @dev See {IERC777-symbol}.
87:      */
88:     function symbol() public view virtual override returns (string memory) {
89:         return _symbol;
90:     }
91:
92:     /**
93:      * @dev See {ERC20-decimals}.
94:      *
95:      * Always returns 18, as per the
96:      * [ERC777 EIP](https://eips.ethereum.org/EIPS/eip-777#backward-compatibility).
97:      */
98:     function decimals() public pure virtual returns (uint8) {
99:         return 18;
100:     }
101:
102:     /**
103:      * @dev See {IERC777-granularity}.
104:      *
105:      * This implementation always returns `1`.
106:      */
107:     function granularity() public view virtual override returns (uint256) {
108:         return 1;
109:     }
110:
111:     /**
112:      * @dev See {IERC777-totalSupply}.
113:      */
114:     function totalSupply() public view virtual override(IERC20, IERC777) returns (uint256) {
115:         return _totalSupply;
116:     }
117:
118:     /**
119:      * @dev Returns the amount of tokens owned by an account (`tokenHolder`).
120:      */
121:     function balanceOf(address tokenHolder) public view virtual override(IERC20, IERC777) returns (uint256) {
122:         return _balances[tokenHolder];
123:     }
124:
125:     /**
126:      * @dev See {IERC777-send}.
127:      *
128:      * Also emits a {IERC20-Transfer} event for ERC20 compatibility.
129:      */
130:     function send(
131:         address recipient,
132:         uint256 amount,
133:         bytes memory data
134:     ) public virtual override {
135:         _send(_msgSender(), recipient, amount, data, "", true);
136:     }
137:
138:     /**
139:      * @dev See {IERC20-transfer}.
140:      *
```

```
     * Unlike `send`, `recipient` is _not_ required to implement the {IERC777Recipient}
     * interface if it is a contract.
     *
     * Also emits a {Sent} event.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        require(recipient != address(0), "ERC777: transfer to the zero address");

        address from = _msgSender();

        _callTokensToSend(from, from, recipient, amount, "", "");

        _move(from, from, recipient, amount, "", "");

        _callTokensReceived(from, from, recipient, amount, "", "", false);

        return true;
    }

    /**
     * @dev See {IERC777-burn}.
     *
     * Also emits a {IERC20-Transfer} event for ERC20 compatibility.
     */
    function burn(uint256 amount, bytes memory data) public virtual override {
        _burn(_msgSender(), amount, data, "");
    }

    /**
     * @dev See {IERC777-isOperatorFor}.
     */
    function isOperatorFor(address operator, address tokenHolder) public view virtual override returns (bool) {
        return
            operator == tokenHolder ||
            (_defaultOperators[operator] && !_revokedDefaultOperators[tokenHolder][operator]) ||
            _operators[tokenHolder][operator];
    }

    /**
     * @dev See {IERC777-authorizeOperator}.
     */
    function authorizeOperator(address operator) public virtual override {
        require(_msgSender() != operator, "ERC777: authorizing self as operator");

        if (_defaultOperators[operator]) {
            delete _revokedDefaultOperators[_msgSender()][operator];
        } else {
            _operators[_msgSender()][operator] = true;
        }

        emit AuthorizedOperator(operator, _msgSender());
    }

    /**
     * @dev See {IERC777-revokeOperator}.
     */
    function revokeOperator(address operator) public virtual override {
        require(operator != _msgSender(), "ERC777: revoking self as operator");

        if (_defaultOperators[operator]) {
            _revokedDefaultOperators[_msgSender()][operator] = true;
        } else {
            delete _operators[_msgSender()][operator];
        }

        emit RevokedOperator(operator, _msgSender());
    }

    /**
     * @dev See {IERC777-defaultOperators}.
     */
    function defaultOperators() public view virtual override returns (address[] memory) {
```

```
213:         return _defaultOperatorsArray;
214:     }
215:
216:     /**
217:      * @dev See {IERC777-operatorSend}.
218:      *
219:      * Emits {Sent} and {IERC20-Transfer} events.
220:      */
221:     function operatorSend(
222:         address sender,
223:         address recipient,
224:         uint256 amount,
225:         bytes memory data,
226:         bytes memory operatorData
227:     ) public virtual override {
228:         require(isOperatorFor(_msgSender(), sender), "ERC777: caller is not an operator for holder");
229:         _send(sender, recipient, amount, data, operatorData, true);
230:     }
231:
232:     /**
233:      * @dev See {IERC777-operatorBurn}.
234:      *
235:      * Emits {Burned} and {IERC20-Transfer} events.
236:      */
237:     function operatorBurn(
238:         address account,
239:         uint256 amount,
240:         bytes memory data,
241:         bytes memory operatorData
242:     ) public virtual override {
243:         require(isOperatorFor(_msgSender(), account), "ERC777: caller is not an operator for holder");
244:         _burn(account, amount, data, operatorData);
245:     }
246:
247:     /**
248:      * @dev See {IERC20-allowance}.
249:      *
250:      * Note that operator and allowance concepts are orthogonal: operators may
251:      * not have allowance, and accounts with allowance may not be operators
252:      * themselves.
253:      */
254:     function allowance(address holder, address spender) public view virtual override returns (uint256) {
255:         return _allowances[holder][spender];
256:     }
257:
258:     /**
259:      * @dev See {IERC20-approve}.
260:      *
261:      * NOTE: If `value` is the maximum `uint256`, the allowance is not updated on
262:      * `transferFrom`. This is semantically equivalent to an infinite approval.
263:      *
264:      * Note that accounts cannot have allowance issued by their operators.
265:      */
266:     function approve(address spender, uint256 value) public virtual override returns (bool) {
267:         address holder = _msgSender();
268:         _approve(holder, spender, value);
269:         return true;
270:     }
271:
272:     /**
273:      * @dev See {IERC20-transferFrom}.
274:      *
275:      * NOTE: Does not update the allowance if the current allowance
276:      * is the maximum `uint256`.
277:      *
278:      * Note that operator and allowance concepts are orthogonal: operators cannot
279:      * call `transferFrom` (unless they have allowance), and accounts with
280:      * allowance cannot call `operatorSend` (unless they are operators).
281:      *
282:      * Emits {Sent}, {IERC20-Transfer} and {IERC20-Approval} events.
283:      */
284:     function transferFrom(
```

```
285:         address holder,
286:         address recipient,
287:         uint256 amount
288:     ) public virtual override returns (bool) {
289:         require(recipient != address(0), "ERC777: transfer to the zero address");
290:         require(holder != address(0), "ERC777: transfer from the zero address");
291:
292:         address spender = _msgSender();
293:
294:         _callTokensToSend(spender, holder, recipient, amount, "", "");
295:
296:         _spendAllowance(holder, spender, amount);
297:
298:         _move(spender, holder, recipient, amount, "", "");
299:
300:         _callTokensReceived(spender, holder, recipient, amount, "", "", false);
301:
302:         return true;
303:     }
304:
305:     /**
306:      * @dev Creates `amount` tokens and assigns them to `account`, increasing
307:      * the total supply.
308:      *
309:      * If a send hook is registered for `account`, the corresponding function
310:      * will be called with `operator`, `data` and `operatorData`.
311:      *
312:      * See {IERC777Sender} and {IERC777Recipient}.
313:      *
314:      * Emits {Minted} and {IERC20-Transfer} events.
315:      *
316:      * Requirements
317:      *
318:      * - `account` cannot be the zero address.
319:      * - if `account` is a contract, it must implement the {IERC777Recipient}
320:      * interface.
321:      */
322:     function _mint(
323:         address account,
324:         uint256 amount,
325:         bytes memory userData,
326:         bytes memory operatorData
327:     ) internal virtual {
328:         _mint(account, amount, userData, operatorData, true);
329:     }
330:
331:     /**
332:      * @dev Creates `amount` tokens and assigns them to `account`, increasing
333:      * the total supply.
334:      *
335:      * If `requireReceptionAck` is set to true, and if a send hook is
336:      * registered for `account`, the corresponding function will be called with
337:      * `operator`, `data` and `operatorData`.
338:      *
339:      * See {IERC777Sender} and {IERC777Recipient}.
340:      *
341:      * Emits {Minted} and {IERC20-Transfer} events.
342:      *
343:      * Requirements
344:      *
345:      * - `account` cannot be the zero address.
346:      * - if `account` is a contract, it must implement the {IERC777Recipient}
347:      * interface.
348:      */
349:     function _mint(
350:         address account,
351:         uint256 amount,
352:         bytes memory userData,
353:         bytes memory operatorData,
354:         bool requireReceptionAck
355:     ) internal virtual {
356:         require(account != address(0), "ERC777: mint to the zero address");
```

```
357:
358:         address operator = _msgSender();
359:
360:         _beforeTokenTransfer(operator, address(0), account, amount);
361:
362:         // Update state variables
363:         _totalSupply += amount;
364:         _balances[account] += amount;
365:
366:         _callTokensReceived(operator, address(0), account, amount, userData, operatorData, requireReceptionAck);
367:
368:         emit Minted(operator, account, amount, userData, operatorData);
369:         emit Transfer(address(0), account, amount);
370:     }
371:
372:     /**
373:      * @dev Send tokens
374:      * @param from address token holder address
375:      * @param to address recipient address
376:      * @param amount uint256 amount of tokens to transfer
377:      * @param userData bytes extra information provided by the token holder (if any)
378:      * @param operatorData bytes extra information provided by the operator (if any)
379:      * @param requireReceptionAck if true, contract recipients are required to implement ERC777TokensRecipient
380:      */
381:     function _send(
382:         address from,
383:         address to,
384:         uint256 amount,
385:         bytes memory userData,
386:         bytes memory operatorData,
387:         bool requireReceptionAck
388:     ) internal virtual {
389:         require(from != address(0), "ERC777: send from the zero address");
390:         require(to != address(0), "ERC777: send to the zero address");
391:
392:         address operator = _msgSender();
393:
394:         _callTokensToSend(operator, from, to, amount, userData, operatorData);
395:
396:         _move(operator, from, to, amount, userData, operatorData);
397:
398:         _callTokensReceived(operator, from, to, amount, userData, operatorData, requireReceptionAck);
399:     }
400:
401:     /**
402:      * @dev Burn tokens
403:      * @param from address token holder address
404:      * @param amount uint256 amount of tokens to burn
405:      * @param data bytes extra information provided by the token holder
406:      * @param operatorData bytes extra information provided by the operator (if any)
407:      */
408:     function _burn(
409:         address from,
410:         uint256 amount,
411:         bytes memory data,
412:         bytes memory operatorData
413:     ) internal virtual {
414:         require(from != address(0), "ERC777: burn from the zero address");
415:
416:         address operator = _msgSender();
417:
418:         _callTokensToSend(operator, from, address(0), amount, data, operatorData);
419:
420:         _beforeTokenTransfer(operator, from, address(0), amount);
421:
422:         // Update state variables
423:         uint256 fromBalance = _balances[from];
424:         require(fromBalance >= amount, "ERC777: burn amount exceeds balance");
425:         unchecked {
426:             _balances[from] = fromBalance - amount;
427:         }
428:         _totalSupply -= amount;
```

```
429:
430:         emit Burned(operator, from, amount, data, operatorData);
431:         emit Transfer(from, address(0), amount);
432:     }
433:
434:     function _move(
435:         address operator,
436:         address from,
437:         address to,
438:         uint256 amount,
439:         bytes memory userData,
440:         bytes memory operatorData
441:     ) private {
442:         _beforeTokenTransfer(operator, from, to, amount);
443:
444:         uint256 fromBalance = _balances[from];
445:         require(fromBalance >= amount, "ERC777: transfer amount exceeds balance");
446:         unchecked {
447:             _balances[from] = fromBalance - amount;
448:         }
449:         _balances[to] += amount;
450:
451:         emit Sent(operator, from, to, amount, userData, operatorData);
452:         emit Transfer(from, to, amount);
453:     }
454:
455:     /**
456:      * @dev See {ERC20-_approve}.
457:      *
458:      * Note that accounts cannot have allowance issued by their operators.
459:      */
460:     function _approve(
461:         address holder,
462:         address spender,
463:         uint256 value
464:     ) internal virtual {
465:         require(holder != address(0), "ERC777: approve from the zero address");
466:         require(spender != address(0), "ERC777: approve to the zero address");
467:
468:         _allowances[holder][spender] = value;
469:         emit Approval(holder, spender, value);
470:     }
471:
472:     /**
473:      * @dev Call from.tokensToSend() if the interface is registered
474:      * @param operator address operator requesting the transfer
475:      * @param from address token holder address
476:      * @param to address recipient address
477:      * @param amount uint256 amount of tokens to transfer
478:      * @param userData bytes extra information provided by the token holder (if any)
479:      * @param operatorData bytes extra information provided by the operator (if any)
480:      */
481:     function _callTokensToSend(
482:         address operator,
483:         address from,
484:         address to,
485:         uint256 amount,
486:         bytes memory userData,
487:         bytes memory operatorData
488:     ) private {
489:         address implementer = _ERC1820_REGISTRY.getInterfaceImplementer(from, _TOKENS_SENDER_INTERFACE_HASH);
490:         if (implementer != address(0)) {
491:             IERC777Sender(implementer).tokensToSend(operator, from, to, amount, userData, operatorData);
492:         }
493:     }
494:
495:     /**
496:      * @dev Call to.tokensReceived() if the interface is registered. Reverts if the recipient is a contract but
497:      * tokensReceived() was not registered for the recipient
498:      * @param operator address operator requesting the transfer
499:      * @param from address token holder address
500:      * @param to address recipient address
```

```
501:      * @param amount uint256 amount of tokens to transfer
502:      * @param userData bytes extra information provided by the token holder (if any)
503:      * @param operatorData bytes extra information provided by the operator (if any)
504:      * @param requireReceptionAck if true, contract recipients are required to implement ERC777TokensRecipient
505:      */
506:     function _callTokensReceived(
507:         address operator,
508:         address from,
509:         address to,
510:         uint256 amount,
511:         bytes memory userData,
512:         bytes memory operatorData,
513:         bool requireReceptionAck
514:     ) private {
515:         address implementer = _ERC1820_REGISTRY.getInterfaceImplementer(to, _TOKENS_RECIPIENT_INTERFACE_HASH);
516:         if (implementer != address(0)) {
517:             IERC777Recipient(implementer).tokensReceived(operator, from, to, amount, userData, operatorData);
518:         } else if (requireReceptionAck) {
519:             require(!to.isContract(), "ERC777: token recipient contract has no implementer for ERC777TokensRecipient");
520:         }
521:     }
522:
523:     /**
524:      * @dev Spend `amount` form the allowance of `owner` toward `spender`.
525:      *
526:      * Does not update the allowance amount in case of infinite allowance.
527:      * Revert if not enough allowance is available.
528:      *
529:      * Might emit an {Approval} event.
530:      */
531:     function _spendAllowance(
532:         address owner,
533:         address spender,
534:         uint256 amount
535:     ) internal virtual {
536:         uint256 currentAllowance = allowance(owner, spender);
537:         if (currentAllowance != type(uint256).max) {
538:             require(currentAllowance >= amount, "ERC777: insufficient allowance");
539:             unchecked {
540:                 _approve(owner, spender, currentAllowance - amount);
541:             }
542:         }
543:     }
544:
545:     /**
546:      * @dev Hook that is called before any token transfer. This includes
547:      * calls to {send}, {transfer}, {operatorSend}, minting and burning.
548:      *
549:      * Calling conditions:
550:      *
551:      * - when `from` and `to` are both non-zero, `amount` of ``from``'s tokens
552:      * will be to transferred to `to`.
553:      * - when `from` is zero, `amount` tokens will be minted for `to`.
554:      * - when `to` is zero, `amount` of ``from``'s tokens will be burned.
555:      * - `from` and `to` are never both zero.
556:      *
557:      * To learn more about hooks, head to xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
558:      */
559:     function _beforeTokenTransfer(
560:         address operator,
561:         address from,
562:         address to,
563:         uint256 amount
564:     ) internal virtual {}
565: }
```

start with:

```
contract ERC777 is Context, IERC777, IERC20 {
```

The contract inherits from the IERC777 interface and the IERC20 interface. This means that it will have to support both interfaces specifications.

Interfaces are defined in ERC1820 and require that contracts publish a standard interface specification to the interface registery.

The constructor

```
    constructor(
        string memory name_,
        string memory symbol_,
        address[] memory defaultOperators_
    ) {
        _name = name_;
        _symbol = symbol_;

        _defaultOperatorsArray = defaultOperators_;
        for (uint256 i = 0; i < defaultOperators_.length; i++) {
            _defaultOperators[defaultOperators_[i]] = true;
        }

        // register interfaces
        _ERC1820_REGISTRY.setInterfaceImplementer(address(this), keccak256("ERC777Token"), address(this));
        _ERC1820_REGISTRY.setInterfaceImplementer(address(this), keccak256("ERC20Token"), address(this));
    }
```

Takes a list of defaultOperators that may be empty.

At the bottom of the contract the gurantees of matching with the registry are setup. The constructor will not succede if it fails to match the standard.

The symbol is a short name, `S7` , and the name is `Simple777Token` .

Send

```
/**
 * @dev Send tokens
 * @param from address token holder address
 * @param to address recipient address
 * @param amount uint256 amount of tokens to transfer
 * @param userData bytes extra information provided by the token holder (if any)
 * @param operatorData bytes extra information provided by the operator (if any)
 * @param requireReceptionAck if true, contract recipients are required to implement ERC777TokensRecipient
 */
function _send(
    address from,
    address to,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData,
    bool requireReceptionAck
) internal virtual {
    require(from != address(0), "ERC777: send from the zero address");
    require(to != address(0), "ERC777: send to the zero address");

    address operator = _msgSender();

    _callTokensToSend(operator, from, to, amount, userData, operatorData);

    _move(operator, from, to, amount, userData, operatorData);

    _callTokensReceived(operator, from, to, amount, userData, operatorData, requireReceptionAck);
}
```

Send has basic checks that you are not sending to or receiving from a 0 address.

It calls the before hook, moves the tokens, then calls the after hook. The work is in the `_move` call.

```
function _move(
    address operator,
    address from,
    address to,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData
) private {
    _beforeTokenTransfer(operator, from, to, amount);

    uint256 fromBalance = _balances[from];
    require(fromBalance >= amount, "ERC777: transfer amount exceeds balance");
    unchecked {
        _balances[from] = fromBalance - amount;
    }
    _balances[to] += amount;

    emit Sent(operator, from, to, amount, userData, operatorData);
    emit Transfer(from, to, amount);
}
```

An overidable function `_beforeTokenTransfer` is called. You can replace this in the derived contract.

It then checks the balance of the sender - that they have enough to send.

`unchecked` is a substraction without overflow chekcing. We have already done the check. The funds are moved.

Then it emits 2 events. Transfer matches with the ERC20 transfer event. Sent is a new event with ERC777.

Send Hook

```
    /**
     * @dev Call from.tokensToSend() if the interface is registered
     * @param operator address operator requesting the transfer
     * @param from address token holder address
     * @param to address recipient address
     * @param amount uint256 amount of tokens to transfer
     * @param userData bytes extra information provided by the token holder (if any)
     * @param operatorData bytes extra information provided by the operator (if any)
     */
    function _callTokensToSend(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes memory userData,
        bytes memory operatorData
    ) private {
        address implementer = _ERC1820_REGISTRY.getInterfaceImplementer(from, _TOKENS_SENDER_INTERFACE_HASH);
        if (implementer != address(0)) {
            IERC777Sender(implementer).tokensToSend(operator, from, to, amount, userData, operatorData);
        }
    }
```

From the IERC777Sender interface specification:

```
interface IERC777Sender {
    /**
     * @dev Called by an {IERC777} token contract whenever a registered holder's
     * (`from`) tokens are about to be moved or destroyed. The type of operation
     * is conveyed by `to` being the zero address or not.
     *
     * This call occurs _before_ the token contract's state is updated, so
     * {IERC777-balanceOf}, etc., can be used to query the pre-operation state.
     *
     * This function may revert to prevent the operation from being executed.
     */
    function tokensToSend(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata userData,
        bytes calldata operatorData
    ) external;
}
```

This looks in the sender to see if they have implemented this interface. If they have then the function is called. If the function fails it will revert the transaction.

The sender can perform tasks like lookup to see if this is a valid operation.

Receive Hook

```
    /**
     * @dev Call to.tokensReceived() if the interface is registered. Reverts if the recipient is a contract but
     * tokensReceived() was not registered for the recipient
     * @param operator address operator requesting the transfer
     * @param from address token holder address
     * @param to address recipient address
     * @param amount uint256 amount of tokens to transfer
     * @param userData bytes extra information provided by the token holder (if any)
     * @param operatorData bytes extra information provided by the operator (if any)
     * @param requireReceptionAck if true, contract recipients are required to implement ERC777TokensRecipient
     */
    function _callTokensReceived(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes memory userData,
        bytes memory operatorData,
        bool requireReceptionAck
    ) private {
        address implementer = _ERC1820_REGISTRY.getInterfaceImplementer(to, _TOKENS_RECIPIENT_INTERFACE_HASH);
        if (implementer != address(0)) {
            IERC777Recipient(implementer).tokensReceived(operator, from, to, amount, userData, operatorData);
        } else if (requireReceptionAck) {
            require(!to.isContract(), "ERC777: token recipient contract has no implementer for ERC777TokensRecipient");
        }
    }
```

And the interface that can be implemented to receive tokens:

```
interface IERC777Recipient {
    /**
     * @dev Called by an {IERC777} token contract whenever tokens are being
     * moved or created into a registered account (`to`). The type of operation
     * is conveyed by `from` being the zero address or not.
     *
     * This call occurs _after_ the token contract's state is updated, so
     * {IERC777-balanceOf}, etc., can be used to query the post-operation state.
     *
     * This function may revert to prevent the operation from being executed.
     */
    function tokensReceived(
        address operator,
        address from,
        address to,
        uint256 amount,
        bytes calldata userData,
        bytes calldata operatorData
    ) external;
}
```

This allows for things like notification of a receiver and for blocking receipt of tokens.

Mint

```
function _mint(
    address account,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData,
    bool requireReceptionAck
) internal virtual {
    require(account != address(0), "ERC777: mint to the zero address");

    address operator = _msgSender();

    _beforeTokenTransfer(operator, address(0), account, amount);

    // Update state variables
    _totalSupply += amount;
    _balances[account] += amount;

    _callTokensReceived(operator, address(0), account, amount, userData, operatorData, requireReceptionAck);

    emit Minted(operator, account, amount, userData, operatorData);
    emit Transfer(address(0), account, amount);
}
```

This is the heart of the process.

Note that this is an `interal virtual` - this means that you can override the mint process with your own function if you need to.

The receiver of the minted tokens gets notified that they have recived them. This is called from the constructor but can also be called from a 'mint' function if that is to be a part of your allowed behavior.

You can create:

```
function mint(
    address account,
    uint256 amount,
    bytes memory userData,
    bytes memory operatorData
) public onlyOwner {
```

for example and only allow the owner to mint or use corresponding security to allow all authorized users to mint - or only a set of "minters".

Burn

```
    /**
     * @dev Burn tokens
     * @param from address token holder address
     * @param amount uint256 amount of tokens to burn
     * @param data bytes extra information provided by the token holder
     * @param operatorData bytes extra information provided by the operator (if any)
     */
    function _burn(
        address from,
        uint256 amount,
        bytes memory data,
        bytes memory operatorData
    ) internal virtual {
        require(from != address(0), "ERC777: burn from the zero address");

        address operator = _msgSender();

        _callTokensToSend(operator, from, address(0), amount, data, operatorData);

        _beforeTokenTransfer(operator, from, address(0), amount);

        // Update state variables
        uint256 fromBalance = _balances[from];
        require(fromBalance >= amount, "ERC777: burn amount exceeds balance");
        unchecked {
            _balances[from] = fromBalance - amount;
        }
        _totalSupply -= amount;

        emit Burned(operator, from, amount, data, operatorData);
        emit Transfer(from, address(0), amount);
    }
```

This should look familiar. This is the inverse operation of mint.

There are already operatorSend and operatorBurn functions.

ERC20 backward compatibility functions

```
    /**
     * @dev See {IERC20-transfer}.
     *
     * Unlike `send`, `recipient` is _not_ required to implement the {IERC777Recipient}
     * interface if it is a contract.
     *
     * Also emits a {Sent} event.
     */
    function transfer(address recipient, uint256 amount) public virtual override returns (bool) {
        require(recipient != address(0), "ERC777: transfer to the zero address");

        address from = _msgSender();

        _callTokensToSend(from, from, recipient, amount, "", "");

        _move(from, from, recipient, amount, "", "");

        _callTokensReceived(from, from, recipient, amount, "", "", false);

        return true;
    }

// ...
```

```
/**
 * @dev See {IERC20-allowance}.
 *
 * Note that operator and allowance concepts are orthogonal: operators may
 * not have allowance, and accounts with allowance may not be operators
 * themselves.
 */
function allowance(address holder, address spender) public view virtual override returns (uint256) {
    return _allowances[holder][spender];
}

/**
 * @dev See {IERC20-approve}.
 *
 * NOTE: If `value` is the maximum `uint256`, the allowance is not updated on
 * `transferFrom`. This is semantically equivalent to an infinite approval.
 *
 * Note that accounts cannot have allowance issued by their operators.
 */
function approve(address spender, uint256 value) public virtual override returns (bool) {
    address holder = _msgSender();
    _approve(holder, spender, value);
    return true;

}

/**
 * @dev See {IERC20-transferFrom}.
 *
 * NOTE: Does not update the allowance if the current allowance
 * is the maximum `uint256`.
 *
 * Note that operator and allowance concepts are orthogonal: operators cannot
 * call `transferFrom` (unless they have allowance), and accounts with
 * allowance cannot call `operatorSend` (unless they are operators).
 *
 * Emits {Sent}, {IERC20-Transfer} and {IERC20-Approval} events.
 */
function transferFrom(
    address holder,
    address recipient,
    uint256 amount
) public virtual override returns (bool) {
    require(recipient != address(0), "ERC777: transfer to the zero address");
    require(holder != address(0), "ERC777: transfer from the zero address");

    address spender = _msgSender();

    _callTokensToSend(spender, holder, recipient, amount, "", "");

    _spendAllowance(holder, spender, amount);

    _move(spender, holder, recipient, amount, "", "");

    _callTokensReceived(spender, holder, recipient, amount, "", "", false);

    return true;
}
```