```solidity
// SPDX-License-Identifier: MIT
// OpenZeppelin Contracts (last updated v4.8.0) (token/ERC721/ERC721.sol)

pragma solidity ^0.8.0;

import "./IERC721.sol";
import "./IERC721Receiver.sol";
import "./extensions/IERC721Metadata.sol";
import "../../utils/Address.sol";
import "../../utils/Context.sol";
import "../../utils/Strings.sol";
import "../../utils/introspection/ERC165.sol";

/**
 * @dev Implementation of https://eips.ethereum.org/EIPS/eip-721[ERC721] Non-Fungible Token Standard, including
 * the Metadata extension, but not including the Enumerable extension, which is available separately as
 * {ERC721Enumerable}.
 */
contract ERC721 is Context, ERC165, IERC721, IERC721Metadata {
    using Address for address;
    using Strings for uint256;

    // Token name
    string private _name;

    // Token symbol
    string private _symbol;

    // Mapping from token ID to owner address
    mapping(uint256 => address) private _owners;

    // Mapping owner address to token count
    mapping(address => uint256) private _balances;

    // Mapping from token ID to approved address
    mapping(uint256 => address) private _tokenApprovals;

    // Mapping from owner to operator approvals
    mapping(address => mapping(address => bool)) private _operatorApprovals;

    /**
     * @dev Initializes the contract by setting a `name` and a `symbol` to the token collection.
     */
    constructor(string memory name_, string memory symbol_) {
        _name = name_;
        _symbol = symbol_;
    }

    /**
     * @dev See {IERC165-supportsInterface}.
     */
    function supportsInterface(bytes4 interfaceId) public view virtual override(ERC165, IERC165) returns (bool) {
        return
            interfaceId == type(IERC721).interfaceId ||
            interfaceId == type(IERC721Metadata).interfaceId ||
            super.supportsInterface(interfaceId);
    }

    /**
     * @dev See {IERC721-balanceOf}.
     */
    function balanceOf(address owner) public view virtual override returns (uint256) {
        require(owner != address(0), "ERC721: address zero is not a valid owner");
        return _balances[owner];
    }

    /**
     * @dev See {IERC721-ownerOf}.
     */
```

```
70      function ownerOf(uint256 tokenId) public view virtual override returns (address) {
71          address owner = _ownerOf(tokenId);
72          require(owner != address(0), "ERC721: invalid token ID");
73          return owner;
74      }
75
76      /**
77       * @dev See {IERC721Metadata-name}.
78       */
79      function name() public view virtual override returns (string memory) {
80          return _name;
81      }
82
83      /**
84       * @dev See {IERC721Metadata-symbol}.
85       */
86      function symbol() public view virtual override returns (string memory) {
87          return _symbol;
88      }
89
90      /**
91       * @dev See {IERC721Metadata-tokenURI}.
92       */
93      function tokenURI(uint256 tokenId) public view virtual override returns (string
        memory) {
94          _requireMinted(tokenId);
95
96          string memory baseURI = _baseURI();
97          return bytes(baseURI).length > 0 ? string(abi.encodePacked(baseURI, tokenId.
            toString())) : "";
98      }
99
100     /**
101      * @dev Base URI for computing {tokenURI}. If set, the resulting URI for each
102      * token will be the concatenation of the `baseURI` and the `tokenId`. Empty
103      * by default, can be overridden in child contracts.
104      */
105     function _baseURI() internal view virtual returns (string memory) {
106         return "";
107     }
108
109     /**
110      * @dev See {IERC721-approve}.
111      */
112     function approve(address to, uint256 tokenId) public virtual override {
113         address owner = ERC721.ownerOf(tokenId);
114         require(to != owner, "ERC721: approval to current owner");
115
116         require(
117             _msgSender() == owner || isApprovedForAll(owner, _msgSender()),
118             "ERC721: approve caller is not token owner or approved for all"
119         );
120
121         _approve(to, tokenId);
122     }
123
124     /**
125      * @dev See {IERC721-getApproved}.
126      */
127     function getApproved(uint256 tokenId) public view virtual override returns (
        address) {
128         _requireMinted(tokenId);
129
130         return _tokenApprovals[tokenId];
131     }
132
133     /**
134      * @dev See {IERC721-setApprovalForAll}.
135      */
136     function setApprovalForAll(address operator, bool approved) public virtual
        override {
137         _setApprovalForAll(_msgSender(), operator, approved);
138     }
```

```
139
140         /**
141          * @dev See {IERC721-isApprovedForAll}.
142          */
143         function isApprovedForAll(address owner, address operator) public view virtual
            override returns (bool) {
144             return _operatorApprovals[owner][operator];
145         }
146
147         /**
148          * @dev See {IERC721-transferFrom}.
149          */
150         function transferFrom(
151             address from,
152             address to,
153             uint256 tokenId
154         ) public virtual override {
155             //solhint-disable-next-line max-line-length
156             require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: caller is not
                token owner or approved");
157
158             _transfer(from, to, tokenId);
159         }
160
161         /**
162          * @dev See {IERC721-safeTransferFrom}.
163          */
164         function safeTransferFrom(
165             address from,
166             address to,
167             uint256 tokenId
168         ) public virtual override {
169             safeTransferFrom(from, to, tokenId, "");
170         }
171
172         /**
173          * @dev See {IERC721-safeTransferFrom}.
174          */
175         function safeTransferFrom(
176             address from,
177             address to,
178             uint256 tokenId,
179             bytes memory data
180         ) public virtual override {
181             require(_isApprovedOrOwner(_msgSender(), tokenId), "ERC721: caller is not
                token owner or approved");
182             _safeTransfer(from, to, tokenId, data);
183         }
184
185         /**
186          * @dev Safely transfers `tokenId` token from `from` to `to`, checking first that
            contract recipients
187          * are aware of the ERC721 protocol to prevent tokens from being forever locked.
188          *
189          * `data` is additional data, it has no specified format and it is sent in call
            to `to`.
190          *
191          * This internal function is equivalent to {safeTransferFrom}, and can be used to
            e.g.
192          * implement alternative mechanisms to perform token transfer, such as
            signature-based.
193          *
194          * Requirements:
195          *
196          * - `from` cannot be the zero address.
197          * - `to` cannot be the zero address.
198          * - `tokenId` token must exist and be owned by `from`.
199          * - If `to` refers to a smart contract, it must implement
            {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
200          *
201          * Emits a {Transfer} event.
202          */
203         function _safeTransfer(
```

```
204             address from,
205             address to,
206             uint256 tokenId,
207             bytes memory data
208         ) internal virtual {
209             _transfer(from, to, tokenId);
210             require(_checkOnERC721Received(from, to, tokenId, data), "ERC721: transfer to
                non ERC721Receiver implementer");
211         }
212
213         /**
214          * @dev Returns the owner of the `tokenId`. Does NOT revert if token doesn't exist
215          */
216         function _ownerOf(uint256 tokenId) internal view virtual returns (address) {
217             return _owners[tokenId];
218         }
219
220         /**
221          * @dev Returns whether `tokenId` exists.
222          *
223          * Tokens can be managed by their owner or approved accounts via {approve} or
             {setApprovalForAll}.
224          *
225          * Tokens start existing when they are minted (`_mint`),
226          * and stop existing when they are burned (`_burn`).
227          */
228         function _exists(uint256 tokenId) internal view virtual returns (bool) {
229             return _ownerOf(tokenId) != address(0);
230         }
231
232         /**
233          * @dev Returns whether `spender` is allowed to manage `tokenId`.
234          *
235          * Requirements:
236          *
237          * - `tokenId` must exist.
238          */
239         function _isApprovedOrOwner(address spender, uint256 tokenId) internal view
        virtual returns (bool) {
240             address owner = ERC721.ownerOf(tokenId);
241             return (spender == owner || isApprovedForAll(owner, spender) ||
                getApproved(tokenId) == spender);
242         }
243
244         /**
245          * @dev Safely mints `tokenId` and transfers it to `to`.
246          *
247          * Requirements:
248          *
249          * - `tokenId` must not exist.
250          * - If `to` refers to a smart contract, it must implement
             {IERC721Receiver-onERC721Received}, which is called upon a safe transfer.
251          *
252          * Emits a {Transfer} event.
253          */
254         function _safeMint(address to, uint256 tokenId) internal virtual {
255             _safeMint(to, tokenId, "");
256         }
257
258         /**
259          * @dev Same as {xref-ERC721-_safeMint-address-uint256-}[`_safeMint`], with an
             additional `data` parameter which is
260          * forwarded in {IERC721Receiver-onERC721Received} to contract recipients.
261          */
262         function _safeMint(
263             address to,
264             uint256 tokenId,
265             bytes memory data
266         ) internal virtual {
267             _mint(to, tokenId);
268             require(
269                 _checkOnERC721Received(address(0), to, tokenId, data),
270                 "ERC721: transfer to non ERC721Receiver implementer"
```

```
271             );
272         }
273 
274         /**
275          * @dev Mints `tokenId` and transfers it to `to`.
276          *
277          * WARNING: Usage of this method is discouraged, use {_safeMint} whenever possible
278          *
279          * Requirements:
280          *
281          * - `tokenId` must not exist.
282          * - `to` cannot be the zero address.
283          *
284          * Emits a {Transfer} event.
285          */
286         function _mint(address to, uint256 tokenId) internal virtual {
287             require(to != address(0), "ERC721: mint to the zero address");
288             require(!_exists(tokenId), "ERC721: token already minted");
289 
290             _beforeTokenTransfer(address(0), to, tokenId, 1);
291 
292             // Check that tokenId was not minted by `_beforeTokenTransfer` hook
293             require(!_exists(tokenId), "ERC721: token already minted");
294 
295             unchecked {
296                 // Will not overflow unless all 2**256 token ids are minted to the same
                    owner.
297                 // Given that tokens are minted one by one, it is impossible in practice
                    that
298                 // this ever happens. Might change if we allow batch minting.
299                 // The ERC fails to describe this case.
300                 _balances[to] += 1;
301             }
302 
303             _owners[tokenId] = to;
304 
305             emit Transfer(address(0), to, tokenId);
306 
307             _afterTokenTransfer(address(0), to, tokenId, 1);
308         }
309 
310         /**
311          * @dev Destroys `tokenId`.
312          * The approval is cleared when the token is burned.
313          * This is an internal function that does not check if the sender is authorized
            to operate on the token.
314          *
315          * Requirements:
316          *
317          * - `tokenId` must exist.
318          *
319          * Emits a {Transfer} event.
320          */
321         function _burn(uint256 tokenId) internal virtual {
322             address owner = ERC721.ownerOf(tokenId);
323 
324             _beforeTokenTransfer(owner, address(0), tokenId, 1);
325 
326             // Update ownership in case tokenId was transferred by `_beforeTokenTransfer`
                hook
327             owner = ERC721.ownerOf(tokenId);
328 
329             // Clear approvals
330             delete _tokenApprovals[tokenId];
331 
332             unchecked {
333                 // Cannot overflow, as that would require more tokens to be
                    burned/transferred
334                 // out than the owner initially received through minting and transferring
                    in.
335                 _balances[owner] -= 1;
336             }
337             delete _owners[tokenId];
```

```
338
339             emit Transfer(owner, address(0), tokenId);
340
341             _afterTokenTransfer(owner, address(0), tokenId, 1);
342         }
343
344         /**
345          * @dev Transfers `tokenId` from `from` to `to`.
346          *  As opposed to {transferFrom}, this imposes no restrictions on msg.sender.
347          *
348          * Requirements:
349          *
350          * - `to` cannot be the zero address.
351          * - `tokenId` token must be owned by `from`.
352          *
353          * Emits a {Transfer} event.
354          */
355         function _transfer(
356             address from,
357             address to,
358             uint256 tokenId
359         ) internal virtual {
360             require(ERC721.ownerOf(tokenId) == from, "ERC721: transfer from incorrect
                owner");
361             require(to != address(0), "ERC721: transfer to the zero address");
362
363             _beforeTokenTransfer(from, to, tokenId, 1);
364
365             // Check that tokenId was not transferred by `_beforeTokenTransfer` hook
366             require(ERC721.ownerOf(tokenId) == from, "ERC721: transfer from incorrect
                owner");
367
368             // Clear approvals from the previous owner
369             delete _tokenApprovals[tokenId];
370
371             unchecked {
372                 // `_balances[from]` cannot overflow for the same reason as described in
                    `_burn`:
373                 // `from`'s balance is the number of token held, which is at least one
                    before the current
374                 // transfer.
375                 // `_balances[to]` could overflow in the conditions described in `_mint`.
                    That would require
376                 // all 2**256 token ids to be minted, which in practice is impossible.
377                 _balances[from] -= 1;
378                 _balances[to] += 1;
379             }
380             _owners[tokenId] = to;
381
382             emit Transfer(from, to, tokenId);
383
384             _afterTokenTransfer(from, to, tokenId, 1);
385         }
386
387         /**
388          * @dev Approve `to` to operate on `tokenId`
389          *
390          * Emits an {Approval} event.
391          */
392         function _approve(address to, uint256 tokenId) internal virtual {
393             _tokenApprovals[tokenId] = to;
394             emit Approval(ERC721.ownerOf(tokenId), to, tokenId);
395         }
396
397         /**
398          * @dev Approve `operator` to operate on all of `owner` tokens
399          *
400          * Emits an {ApprovalForAll} event.
401          */
402         function _setApprovalForAll(
403             address owner,
404             address operator,
405             bool approved
```

```
406         ) internal virtual {
407             require(owner != operator, "ERC721: approve to caller");
408             _operatorApprovals[owner][operator] = approved;
409             emit ApprovalForAll(owner, operator, approved);
410         }
411 
412         /**
413          * @dev Reverts if the `tokenId` has not been minted yet.
414          */
415         function _requireMinted(uint256 tokenId) internal view virtual {
416             require(_exists(tokenId), "ERC721: invalid token ID");
417         }
418 
419         /**
420          * @dev Internal function to invoke {IERC721Receiver-onERC721Received} on a
             target address.
421          * The call is not executed if the target address is not a contract.
422          *
423          * @param from address representing the previous owner of the given token ID
424          * @param to target address that will receive the tokens
425          * @param tokenId uint256 ID of the token to be transferred
426          * @param data bytes optional data to send along with the call
427          * @return bool whether the call correctly returned the expected magic value
428          */
429         function _checkOnERC721Received(
430             address from,
431             address to,
432             uint256 tokenId,
433             bytes memory data
434         ) private returns (bool) {
435             if (to.isContract()) {
436                 try IERC721Receiver(to).onERC721Received(_msgSender(), from, tokenId, data
                    ) returns (bytes4 retval) {
437                     return retval == IERC721Receiver.onERC721Received.selector;
438                 } catch (bytes memory reason) {
439                     if (reason.length == 0) {
440                         revert("ERC721: transfer to non ERC721Receiver implementer");
441                     } else {
442                         /// @solidity memory-safe-assembly
443                         assembly {
444                             revert(add(32, reason), mload(reason))
445                         }
446                     }
447                 }
448             } else {
449                 return true;
450             }
451         }
452 
453         /**
454          * @dev Hook that is called before any token transfer. This includes minting and
             burning. If {ERC721Consecutive} is
455          * used, the hook may be called as part of a consecutive (batch) mint, as
             indicated by `batchSize` greater than 1.
456          *
457          * Calling conditions:
458          *
459          * - When `from` and `to` are both non-zero, ``from``'s tokens will be
             transferred to `to`.
460          * - When `from` is zero, the tokens will be minted for `to`.
461          * - When `to` is zero, ``from``'s tokens will be burned.
462          * - `from` and `to` are never both zero.
463          * - `batchSize` is non-zero.
464          *
465          * To learn more about hooks, head to
             xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
466          */
467         function _beforeTokenTransfer(
468             address from,
469             address to,
470             uint256, /* firstTokenId */
471             uint256 batchSize
472         ) internal virtual {
```

```
473             if (batchSize > 1) {
474                 if (from != address(0)) {
475                     _balances[from] -= batchSize;
476                 }
477                 if (to != address(0)) {
478                     _balances[to] += batchSize;
479                 }
480             }
481         }
482 
483         /**
484          * @dev Hook that is called after any token transfer. This includes minting and
             burning. If {ERC721Consecutive} is
485          * used, the hook may be called as part of a consecutive (batch) mint, as
             indicated by `batchSize` greater than 1.
486          *
487          * Calling conditions:
488          *
489          * - When `from` and `to` are both non-zero, ``from``'s tokens were transferred
             to `to`.
490          * - When `from` is zero, the tokens were minted for `to`.
491          * - When `to` is zero, ``from``'s tokens were burned.
492          * - `from` and `to` are never both zero.
493          * - `batchSize` is non-zero.
494          *
495          * To learn more about hooks, head to
             xref:ROOT:extending-contracts.adoc#using-hooks[Using Hooks].
496          */
497         function _afterTokenTransfer(
498             address from,
499             address to,
500             uint256 firstTokenId,
501             uint256 batchSize
502         ) internal virtual {}
503     }
504 
```