CatalogContract.sol

```
pragma solidity ^0.4.0;

contract BaseContentManagementContract {
    address public author;
    bytes32 public name = "";
    bytes32 public genre = "";
    function murder() public;
}

contract CatalogContract {

    /* VARIABLES */
    // Constants
    uint public contentCost = 0.01 ether;   // ~ 4€
    uint public premiumCost = 0.1 ether;    // ~ 40€
    uint public premiumTime = 172800;       // ~ 1 month
    uint public payAfter = 10;  // views

    // Runtime
    address public owner;
    uint private balance = 0;

    // Structs
    struct content {
        bytes32 name;
        address author;
        bytes32 genre;
        uint views;
    }

    struct author {
        bool alreadyFound;
        uint views;
        uint uncollectedViews;
    }

    mapping (address => uint) private premiumUsers; // map a user into his
subscription expiration time
    mapping (address => mapping (address => bool)) private
accessibleContent;   // map a user into his accessible contents
    address[] contentsList;  // list of all contents
    mapping (address => content) contents;  // map content addresses into
contents
    address[] authorsList;   // list of all authors
    mapping (address => author) authors;    // map author address in its
struct


    /* EVENTS */
    event FallbackFunctionCall(string message, bytes data);
    event CatalogClosed();
    event grantedAccess(address user, address content);
    event paymentAvailable(address content);
    event becomesPremium(address user);
    event newContentAvailable(bytes32 name, address addr);


    /* MODIFIERS */
    modifier onlyOwner() {
        require(msg.sender == owner,
            "Only the contract owner can perform this action.");
        _;
    }

    modifier exists(address c) {
        require(contents[c].name != "" &&
        BaseContentManagementContract(c).author() != 0);
        _;
    }


    /* FUNCTIONS */

    /** Constructor */
    constructor() public {
        owner = msg.sender;
    }

    /** Fallback function */
    function () public {
        revert();
    }

    /** Suicide function, can be called only by the owner */
    function _suicide() public onlyOwner {
        // Murder all the contents in the catalog: this will free up space
in
        // the blockchain and create negative gas to consume less in this
        // process: all this transfers cost a lot.
        for (uint i = 0; i < contentsList.length; i++) {
            BaseContentManagementContract(contentsList[i]).murder();
        }
        // Distribute the balance to the authors according with their views
        // count
        uint totalViews = 0;
        uint totalUncollectedViews = 0;
        // Calculate totals of views and uncollectedViews of all the authors
        for (i = 0; i < authorsList.length; i++) {
            author memory a = authors[authorsList[i]];
            totalViews += a.views;
            totalUncollectedViews += a.uncollectedViews;
        }
        // subtract from the balance the amount that has to be payed for the
        // uncollected views to the authors
        if (totalViews != 0) {
            balance -= totalUncollectedViews * contentCost;
            for (i = 0; i < authorsList.length; i++) {
                a = authors[authorsList[i]];
                // for each author pay the uncollected views
                uint256 amountFromUncollectedViews = a.uncollectedViews *
contentCost;
                // distribute the remaining balance to the authors according
with
                // their views count
                uint256 amountFromPremium = balance * a.views / totalViews;
                uint256 amount = amountFromUncollectedViews +
amountFromPremium;
                if (amount != 0) authorsList[i].transfer(amount);
            }
        }
        // emit an event
        emit CatalogClosed();
        // should not, but if there is some wei in excess transfer it to the
        // owner
        selfdestruct(owner);
    }

    /** Pays for access to content x.
     * @param x the address of the block of the ContentManagementContract.
     * Gas: who requests the content pays.
     */
    function getContent(address x) public payable exists(x) {
        grantAccess(msg.sender, x);
    }

    /** Requests access to content x without paying, premium accounts only.
     * @param x the address of the block of the ContentManagementContract.
     * Gas: who requests the content pays.
     */
    /* DEPRECATED: a user could pay only a premium cycle and access all
content (at most once) in the future,
     * even if the premium account is no longer active. In this case the
premium account would have turned into a
     * "bundle" of content rather than a subscription. Since this is not the
expected behavior, the function has
     * been abolished and now a premium user can consume all content without
having to first request access to it
     * as long as his premium subscription still valid.
     * Access to content from a premium account will not affect previously
purchased content. The user can still
     * consume the purchased content (once only) when the subscription has
ended, even if that content has been
     * accessed by this user multiple times during his premium account.
     * In addition, a premium account can also purchase content. They will
be consumable (once only) when the
     * premium subscription has ended.
    function getContentPremium(address x) public exists(x) {
        require(isPremium(msg.sender));
        accessibleContent[msg.sender][x] = true;
        emit grantedAccess(x, msg.sender);
    }*/

    /** Pays for granting access to content x to the user u.
     * @param x the address of the block of the ContentManagementContract.
     * @param u the user to whom you want to gift the content.
     * Gas: who gift pays.
     */
    function giftContent(address x, address u) public payable exists(x) {
        grantAccess(u, x);
    }
    /** Pays for granting a Premium Account to the user u.
     * @param u the user to whom you want to gift the subscription.
     * Gas: who gift pays.
     */
    function giftPremium(address u) public payable {
        setPremium(u);
    }

    /** Starts a new premium subscription.
     * Gas: who subscribe pays.
     */
    function buyPremium() public payable {
        setPremium(msg.sender);
    }

    /** Used by the authors to collect their reached payout.
     * The author contents must has been visited at least payAfter times.
     * (the author should have received the event).
     * Gas: the author (who receives money) pays.
     */
    function collectPayout() public {
        uint uncollectedViews = authors[msg.sender].uncollectedViews;
        require(uncollectedViews >= payAfter, "Your contents have not 
received\
    enough views. Please listen for a paymentAvailable event relative\
    to your address.");
        authors[msg.sender].uncollectedViews = 0;
        uint amount = contentCost * uncollectedViews;
        balance -= amount;
        msg.sender.transfer(amount);
    }

    /** Called from a ContentManagementContract, adds the content to the 
catalog.
     * Gas: the author pays.
     */
    function addMe() public {
        BaseContentManagementContract cc =
            BaseContentManagementContract(msg.sender);
        contents[cc] = content(cc.name(), cc.author(), cc.genre(), 0);
        contentsList.push(cc);
        if (!authors[cc.author()].alreadyFound) {
            authors[cc.author()].alreadyFound = true;
            authorsList.push(cc.author());
        }
        emit newContentAvailable(cc.name(), cc);
    }

    /** Notice the catalog that the user u has consumed the content x.
     * @param u the user that consume the content.
     * Gas: the user that consumes the content pays.
     */
    function consumeContent(address u) public exists(msg.sender) {
        // Premium users can consume contents for free and are not 
considered
        // in the count of views
        if (isPremium(u)) return;
        // Only contents can call this function, so the content to be delete
        // is the msg.sender
        delete accessibleContent[u][msg.sender];
        contents[msg.sender].views++;
        address a = contents[msg.sender].author; // perform only one storage
read
        authors[a].views++;
        authors[a].uncollectedViews++;
        /* Notice the author if his contents has enough views.
         * Note that the event is emitted only once, when the number of
views
         * is exactly equal to payAfter: it is not an oversight but a
caution
         * not to spam too much. Can be changed in >= if this contract is
         * deployed in a dedicated blockchain. */
        if (authors[a].uncollectedViews == payAfter) {
            emit paymentAvailable(a);
        }
    }

    /** Called from a ContentManagementContract, removes the content from
the
     * catalog (used by the suicide function).
     * Gas: the author pays.
     */
    function removesMe() public exists(msg.sender) {
        delete contents[msg.sender];
        bool found = false;
        // Search the address in the array
        for (uint i = 0; i < contentsList.length; i++) {
            // lazy if: skip the storage read if found is true
            if (!found && contentsList[i] == msg.sender) {
                found = true;
            }
            if (found && i < contentsList.length - 1) {
                // move all the following items back of 1 position
                contentsList[i] = contentsList[i+1];
            }
        }
        if (found) {
            // and finally delete the last item
            delete contentsList[contentsList.length - 1];
            contentsList.length--;
        }
    }

    /** Returns the number of views for each content.
     * @return (bytes32[], uint[], address[]), names, addresses and views:
     * each content in names is associated with the views number in views
and
     * with its address in addresses.
     * Gas: no one pay.
     * Burden: O(n).
     */
    function getStatistics() public view returns(bytes32[], address[],
uint[]) {
        bytes32[] memory names = new bytes32[](contentsList.length);
        uint[] memory views = new uint[](contentsList.length);
        for (uint i = 0; i < contentsList.length; i++) {
            content memory c = contents[contentsList[i]]; // perform only
one storage read
            names[i] = c.name;
            views[i] = c.views;
        }
        return (names, contentsList, views);
    }

    /** Returns the list of contents without the number of views.
     * @return (string[], address[]) names and addresses: each content in
names
     * is associated with its address in addresses.
     * Gas: no one pay.
     * Burden: O(n).
     */
    function getContentsList() public view returns(bytes32[], address[]) {
        bytes32[] memory names = new bytes32[](contentsList.length);
        for (uint i = 0; i < contentsList.length; i++) {
            names[i] = contents[contentsList[i]].name;
        }
        return (names, contentsList);
    }

    /** Returns the list of x newest contents.
     * @return (string[], address[]) names and addresses ordered from the
     * newest: each content in names is associated with its address in
addresses.
     * Gas: no one pay.
     * Burden: O(x) ~ O(1).
     */
    function getNewContentsList(uint n) public view returns(bytes32[],
address[]) {
        uint listLength = n;
        // If i have less than chartListLength element in the contentsList I
        // have to return contentsList.length elements
        if (contentsList.length < listLength) listLength =
contentsList.length;
        // NOTE: I assume that the latest content is not the last deployed
contract in the blockchain (with the highest
        // block number), but is the last added to the catalog (that ideally
is when is "published").
        bytes32[] memory names = new bytes32[](listLength);
        address[] memory addresses = new address[](listLength);
        for (uint i = 0; i < listLength; i++) {
            // add it in reverse order: the latest first
            address a = contentsList[contentsList.length - 1 - i];
            names[i] = contents[a].name;
            addresses[i] = a;
        }
        return (names, addresses);
    }

    /** Get the latest release of genre g.
     * @param g the genre of which you want to get the latest content.
     * @return (bytes32, address) names and addresses of the content.
     * Gas: no one pay.
     * Burden: < O(n).
     */
    function getLatestByGenre(bytes32 g) public view returns(bytes32,
address) {
        // using int because i can be negative if the list is empty or there
        // aren't element of genre g. Should not fail.
        int i = int(contentsList.length - 1);
        while (i >= 0)  {
            address addr = contentsList[uint(i)];
            content memory c = contents[addr];
            if (c.genre == g) {
                return (c.name, addr);
            }
            i--;
        }
        // fallback, return empty if not exist a release of g
        return("", 0);
    }

    /** Get most popular release of genre g.
     * @param g the genre of which you want to get the most popular content.
     * @return (string, address) name and address of the most popular
content.
     * If there are 2 or more content with the same number of view the
oldest comes first.
     * Gas: no one pay.
     * Burden: O(n).
     */
    function getMostPopularByGenre(bytes32 g) public view returns(bytes32,
address) {
        int maxViews = -1;
        bytes32 maxName;
        address maxAddress;
        for (uint i = 0; i < contentsList.length; i++) {
            address addr = contentsList[i];
            content memory c = contents[addr];
            if (c.genre == g && int(c.views) > maxViews) {
                maxViews = int(c.views);
                maxName = c.name;
                maxAddress = addr;
            }
        }
        return (maxName, maxAddress);
    }

    /** Get the latest release of the author a.
     * @param a the author of whom you want to get the latest content.
     * @return (bytes32, address) names and addresses of the content.
     * Gas: no one pay.
     * Burden: < O(n).
     */
    function getLatestByAuthor(address a) public view returns(bytes32,
address) {
        // using int because i can be negative if the list is empty or there
        // aren't element of genre g. Should not fail.
        int i = int(contentsList.length - 1);
        while (i >= 0)  {
            address addr = contentsList[uint(i)];
            content memory c = contents[addr];
            if (c.author == a) {
                return (c.name, addr);
            }
            i--;
        }
        // fallback, return empy if not exist a release of a
        return("", 0);
    }

    /** Get the most popular release of the author a.
     * @param a the author of which you want to get the most popular
content.
     * @return (string, address) name and address of the most popular
content.
     * If there are 2 or more content with the same number of view the
oldest comes first.
     * Gas: no one pay.
     * Burden: O(n).
     */
    function getMostPopularByAuthor(address a) public view returns(bytes32,
address) {
        int maxViews = -1;
        bytes32 maxName;
        address maxAddress;
        for (uint i = 0; i < contentsList.length; i++) {
            address addr = contentsList[i];
            content memory c = contents[addr];
            if (c.author == a && int(c.views) > maxViews) {
                maxViews = int(c.views);
                maxName = c.name;
                maxAddress = addr;
            }
        }
        return (maxName, maxAddress);
    }

    /** Checks if a user u has access to a content x.
     * @param u the user of whom you want to check the access right.
     * @param x the content of which you want to check the access right.
     * @return bool true if the user has the access right, false otherwise.
     * Gas: no one pay.
     * Burden: small.
     */
    function hasAccess(address u, address x) public view exists(x)
returns(bool) {
        // lazy or, premium first because we suppose they consume more
content
        // than standard users
        return isPremium(u) || accessibleContent[u][x];
    }

    /** Checks if a user u has an active premium subscription.
     * @param u the user of whom you want to check the premium subscription.
     * @return bool true if the user hold a still valid premium account,
false
     * otherwise.
     * Gas: no one pay.
     * Burden: small.
     */
    function isPremium(address u) public view returns(bool) {
        return premiumUsers[u] >= block.number;
    }


    /* INTERNAL AUXILIARY FUNCTIONS */

    /** Starts a new premium subscription for the user u based on the amount
v.
     * @param u the user.
     */
    function setPremium(address u) private {
        require(msg.value == premiumCost);
        // If the user has never bought premium or the premium subscription
is
        // expired reset the expiration time to now
        if (!isPremium(u)) premiumUsers[u] = block.number;
        // Increment the user expiration time
        // (if he is already premium will be premium longer)
        premiumUsers[u] += premiumTime;
        emit becomesPremium(u);
        balance += msg.value;
    }

    /** Grant access for the content x to the user v.
    * @param u the user.
    * @param x the content.
    */
    function grantAccess(address u, address x) private {
        require(msg.value == contentCost);
        require(!accessibleContent[u][x]);
        accessibleContent[u][x] = true;
        emit grantedAccess(u, x);
        balance += msg.value;
    }
}
```

BaseContentManagementContract.sol

```
pragma solidity ^0.4.0;

contract CatalogContract {
    function hasAccess(address u, address x) public view returns(bool);
    function consumeContent(address u) public;
    function addMe() public;
    function removesMe() public;
}

contract BaseContentManagementContract {

    /* VARIABLES */

    // Runtime
    address public catalog;
    address public author;
    bytes32 public name;
    bytes32 public genre;
    bool private published = false;
    CatalogContract private catalogContract;


    /* EVENTS */
    event FallbackFunctionCall(string message, bytes data);
    event ContentPublished();
    event ContentDeleted();
    event contentConsumed(address user);


    /* MODIFIERS */
    modifier onlyOwner() {
        require(msg.sender == author, "Only the author can perform this 
action.");
        _;
    }

    modifier validAddress(address addr) {
        uint size;
        assembly { size := extcodesize(addr) }
        require(size > 0, "The address is not valid.");
        _;
    }


    /* FUNCTIONS */
    /** Constructor */
    constructor() public {
        author = msg.sender;
    }

    /** Fallback function */
    function () public {
        revert();
    }
    /** Suicide function, can be called only by the owner */
    function _suicide() public onlyOwner {
        // notice the catalog
        catalogContract.removesMe();
        // emit an event
        emit ContentDeleted();
        // if there is some wei send it to the author
        selfdestruct(author);
    }

    /** Suicide function, can be called only by the owner */
    function murder() public validAddress(catalog) {
        require(msg.sender == catalog);
        // emit an event
        emit ContentDeleted();
        // if there is some wei send it to the author
        selfdestruct(author);
    }

    /** Used by the customers to consume this content after requesting the
access.
     * @return the content.
     */
    function consumeContent() public returns(bytes) {
        require(published, "The content is not yet published.");
        require(catalogContract.hasAccess(msg.sender, this), "You must
reserve this content before accessing it. Please contact the catalog.");
        catalogContract.consumeContent(msg.sender);
        emit contentConsumed(msg.sender);
    }

    /** Used by the author to publish the content.
     * @param c the address of the catalog in which publish the content.
     * The author must specify name and content of this contract before
calling this function.
     * Can be called only one time.
     */
    function publish(address c) public onlyOwner validAddress(c) {
        require(!published, "This contract is already published in the
catalog.");
        require(name[0] != 0, "The content name must be set before publish
the content in the catalog.");
        published = true;
        catalog = c;
        catalogContract = CatalogContract(c);
        catalogContract.addMe();
        emit ContentPublished();
    }
}
```

GenericContentManagementContract.sol

```
pragma solidity ^0.4.0;

import "./BaseContentManagementContract.sol";

contract GenericContentManagementContract is BaseContentManagementContract {

    /* MODIFIERS */

    modifier notNull(bytes32 argument) {
        require(argument[0] != 0, "The argument can not be null.");
        _;
    }

    modifier notEmpty(bytes argument) {
        require(argument.length != 0, "The argument can not be empty.");
        _;
    }

    modifier validAddress(address addr) {
        uint size;
        assembly { size := extcodesize(addr) }
        require(size > 0, "The address is not valid.");
        _;
    }


    /* FUNCTIONS */

    /** Used by the author to set the name.
     * Can be called only one time.
     */
    function setName(bytes32 n) public onlyOwner notNull(n) {
        require(name[0] == 0, "The name can not be overwritten. Use the
suicide function to delete this content and create a new one.");
        name = n;
    }

    /** Used by the author to set the genre.
     * Can be called only one time, but its call is not mandatory (the
content can not have a genre).
     */
    function setGenre(bytes32 g) public onlyOwner notNull(g) {
        require(genre[0] == 0, "The name can not be overwritten. Use the
suicide function to delete this content and create a new one.");
        genre = g;
    }
}
```

demo.js

```
const fs = require('fs');
const solc = require('solc');
const Web3 = require('web3');

const provider = "http://localhost:8545";
const genres = ["adventure", "fantasy", "romance", "horror"];
const contentsNumber = 20;

let web3;
let catalogContract;
let ContentContract;
let latestByAuthor0;
let latestByGenre0;
let contentCost;
let gasLimit;

/**
 * Connects to the Ethereum provider specified in the variable "provider".
 * @returns {Promise<Web3>} a Web3 instance.
 */
function connect() {
  return new Promise((resolve, reject) => {
    web3 = new Web3(new Web3.providers.HttpProvider(provider));
    if (!web3.isConnected()) reject("Cannot connect to "+provider+".");
    console.log("\nConnected to Web3: "+web3.version.node+".\n");
    gasLimit = web3.eth.getBlock("latest").gasLimit;
    resolve(web3);
  })
}

/**
 * Compile a contract.
 * @param filename the sol file to be deployed.
 * @returns {Promise<solc.output>} the compiled contract.
 */
function compileContract(filename = "Contract.sol") {
  return new Promise(resolve => {
    // Compile the source code
    const outputContractName = filename+":"+filename.replace(".sol", "");
    const input = fs.readFileSync(filename);
    function findImports(path) {
      return { contents: fs.readFileSync(path).toString() }
    }
    const source = { };
    source[filename] = input.toString();
    const output = solc.compile({ sources: source }, 1, findImports);
    resolve(output.contracts[outputContractName]);
  })
}

/**
 * Deploy on Web3 a contract.
 * @param compiledContract the compiled contract, give in output by the
compileContract.
 * @param address, optional, the address with which deploy the contract. If
```

```
not specify is the first account.
 * @returns {Promise<web3.eth.contract>} the contract instance.
 */
function deployContract(compiledContract, address = web3.eth.accounts[0]) {
  return new Promise((resolve, reject) => {
    const contract =
web3.eth.contract(JSON.parse(compiledContract.interface));
    const options = {
      data: '0x' + compiledContract.bytecode,
      from: address,
      gas: gasLimit
    };
    // Deploy contract instance
    const contractInstance = contract.new(options, (err, res) => {
      if (err) reject(err);
      if (res.address) {
        console.log(" - contract address: " + res.address);
        resolve(contractInstance);
      }
    });
  })
}

/**
 * Compiles and Deploy on Web3 a contract.
 * @param filename the sol file to be deployed.
 * @param address, optional, the address with which deploy the contract. If
not specify is the first account.
 * @returns {Promise<web3.eth.contract>} the contract instance.
 */
function compileAndDeployContract(filename = "Contract.sol", address =
web3.eth.accounts[0]) {
  return compileContract(filename).then(compiledContract =>
deployContract(compiledContract, address));
}

/**
 * Auxiliary function: generates num contracts, that are object with 3
parameters: name, content and genre.
 * @param num the number of contents that you want to generate.
 * @returns Array of contents object.
 */
function generateContents(num = 0) {
  const contents = [];
  for (let i = 0; i < num; i++)
    contents[i] = {
      name: web3.fromUtf8("title"+i),
      genre: web3.fromUtf8(genres[rand(genres.length-1)])
    };
  return contents;
}

/**
 * Auxiliary function: returns the object needed to call contract functions
that modifies the state.
 * @param from, the person that do the transaction.
 * @param value of the transaction, optional. Default is 0.
 * @param gas, max gas that the transaction can use.
 * @returns object, the object needed for the function call.
 */
function getParams(from = web3.eth.accounts[0], value = 0, gas = gasLimit) {
  return {
    from: from,
    gas: gas,
    value: value
  }
}

/**
 * Auxiliary function: deploy num content contracts on the blockchain and
returns it in an array.
 * @param num the number of contents that you want to deploy.
 * @returns {Promise<contract[]>} an array contract instances.
 */
async function deployContentsContract(num) {
  // compile the contract
  const compiledContract = await
compileContract('GenericContentManagementContract.sol');
  ContentContract =
web3.eth.contract(JSON.parse(compiledContract.interface));
  // deploy num empty Contents
  const contentContracts = [];
  for (let i = 0; i < num; i++) {
    const authorIndex = rand(web3.eth.accounts.length - 1);
    await deployContract(compiledContract, web3.eth.accounts[authorIndex])
      .then(contractInstance => {
        contentContracts.push(contractInstance);
        // exclude the last one because it will deleted
        if (i < num - 1 && authorIndex === 0) latestByAuthor0 =
contractInstance;
      });
  }
  // set the name, content and genre of each content
  const contents = generateContents(num);
  for (let i = 0; i < num; i++) {
    const owner = contentContracts[i].author();
    contentContracts[i].setName(contents[i].name, getParams(owner));
    contentContracts[i].setGenre(contents[i].genre, getParams(owner));
    contentContracts[i].publish(catalogContract.address, getParams(owner));
    // exclude the last one because it will deleted
    if (i < num - 1 && contents[i].genre === web3.fromUtf8(genres[0]))
latestByGenre0 = contentContracts[i];
  }
  return contentContracts;
}

/**
 * Auxiliary function: generates a random number.
 * @param to, optional, the last number of the range.
 * @param from, optional, the starting number of the range.
 * @returns number (random).
 */
function rand(to = 1, from = 0) {
  if (from > to) return -1;
  if (from === to) return from;
  return from + Math.floor(Math.random() * (to - from + 1));
}

/**
 * Auxiliary function: parse the content list returned by
catalogContract.getContentsList().
 * @param contentsList, the content list.
 * @returns Array of object with 2 field: name and address.
 */
function parseContentsList(contentsList = ["", ""]) {
  const list = [];
  for (let i = 0; i < contentsList[0].length; i++) {
    list[i] = {
      name: web3.toUtf8(contentsList[0][i]),
      address: contentsList[1][i]
    }
  }
  return list;
}

/**
 * Auxiliary function: parse the statistics list returned by
catalogContract.getStatistics().
 * @param contentsList, the content list.
 * @returns Array of object with 3 field: name, address and views.
 */
function parseStatistics(contentsList = ["", ""]) {
  const list = [];
  for (let i = 0; i < contentsList[0].length; i++) {
    list[i] = {
      name: web3.toUtf8(contentsList[0][i]),
      address: contentsList[1][i],
      views: contentsList[2][i]
    }
  }
  return list;
}

/**
 * Auxiliary function: print the content list.
 * @param contentsList, the content list.
 */
function printContentsList(contentsList = []) {
  for (let i = 0; i < contentsList.length; i++)
    console.log(" - "+contentsList[i].name+": "+contentsList[i].address);
}

/**
 * Auxiliary function: print the statistics.
 * @param contentsList, the content list.
 */
function printStatistics(contentsList = []) {
  for (let i = 0; i < contentsList.length; i++)
    console.log(" - "+contentsList[i].name+": "+contentsList[i].address+" -
"+contentsList[i].views+" views");
}

/**
 * testing the getLatestByGenre function.
 */
function latestByGenreTest() {
  console.log("\nTesting the getLatestByGenre function on the genre
"+genres[0]+".");
  if(!latestByGenre0) throw "There is no content of genre "+genres[0]+". Try
again with more contents.";
  console.log(" - expected: "+web3.toUtf8(latestByGenre0.name())+":
"+latestByGenre0.address);
  const latest = catalogContract.getLatestByGenre(web3.fromUtf8(genres[0]));
  console.log(" - got (should be the same): "+web3.toUtf8(latest[0])+":
"+latest[1]);
}

/**
 * testing the getLatestByAuthor function.
 */
function latestByAuthorTest() {
  console.log("\nTesting the getLatestByAuthor function on the author
"+web3.eth.accounts[0]+".");
  if(!latestByAuthor0) throw "There is no content of author
"+web3.eth.accounts[0]+". Try again with more contents.";
  console.log(" - expected: "+web3.toUtf8(latestByAuthor0.name())+":
"+latestByAuthor0.address);
  const latest = catalogContract.getLatestByAuthor(web3.eth.accounts[0]);
  console.log(" - got (should be the same): "+web3.toUtf8(latest[0])+":
"+latest[1]);
}

/**
 * testing the getMostPopularByGenre function.
 */
function mostPopularByGenreTest() {
  console.log("\nTesting the getMostPopularByGenre function on the genre
"+genres[0]+".");
  const before =
catalogContract.getMostPopularByGenre(web3.fromUtf8(genres[0]));
  console.log(" - before: "+web3.toUtf8(before[0])+": "+before[1]);
  console.log(" - generating 10 views on the
"+web3.toUtf8(latestByGenre0.name())+" content. " +
    "After that this content should be the most popular.");
  const account = web3.eth.accounts[web3.eth.accounts.length - 1];
  for (let i = 0; i < 10; i++) {
    if (!catalogContract.hasAccess(account, latestByGenre0.address))
      grantAccess(latestByGenre0.address, account);
    consumeContent(latestByGenre0.address, account);
  }
  const after =
catalogContract.getMostPopularByGenre(web3.fromUtf8(genres[0]));
  console.log(" - after: "+web3.toUtf8(after[0])+": "+after[1]);
}

/**
 * testing the getMostPopularByAuthor function.
 */
function mostPopularByAuthorTest() {
  console.log("\nTesting the getMostPopularByAuthor function on the author
"+web3.eth.accounts[0]+".");
  const before =
catalogContract.getMostPopularByAuthor(web3.eth.accounts[0]);
  console.log(" - before: "+web3.toUtf8(before[0])+": "+before[1]);
  console.log(" - generating 10 views on the
"+web3.toUtf8(latestByAuthor0.name())+" content. " +
    "After that this content should be the most popular.");
  const account = web3.eth.accounts[web3.eth.accounts.length - 1];
  for (let i = 0; i < 10; i++) {
    if (!catalogContract.hasAccess(account, latestByAuthor0.address))
      grantAccess(latestByAuthor0.address, account);
    consumeContent(latestByAuthor0.address, account);
  }
  const after =
catalogContract.getMostPopularByAuthor(web3.eth.accounts[0]);
  console.log(" - after: "+web3.toUtf8(after[0])+": "+after[1]);
}

/**
 * Auxiliary function: grant to an user the access to a content.
 * @param contentAddress the content.
 * @param user the user.
 */
function grantAccess(contentAddress, user = web3.eth.accounts[0]) {
  if (!contentCost) contentCost = catalogContract.contentCost();
  catalogContract.getContent(contentAddress, getParams(user, contentCost));
}

/**
 * Small test of the functions to buy and gift content.
 * Are needed at least 3 accounts in web3.eth.accounts and at least 3
contents.
 * The first account (web3.eth.accounts[0]) buys the first two contents.
 * Then the second account (web3.eth.accounts[1]) gifts to the first one the
third content.
 * It is also tested the purchase of an already purchased content (title3)
that should raise an error.
 * @param contentsList, the list of all available contents.
 * @returns Array of the contents on which the first account has access.
 */
function grantAccessTest(contentsList = []) {
  console.log("The first account ("+web3.eth.accounts[0]+") buys the first
two contents:");
  grantAccess(contentsList[0].address);
  //catalogContract.getContent(contentsList[0].address,
getParams(web3.eth.accounts[0], value));
  console.log(" - "+contentsList[0].name);
  grantAccess(contentsList[1].address);
  //catalogContract.getContent(contentsList[1].address,
getParams(web3.eth.accounts[0], value));
  console.log(" - "+contentsList[1].name);
  console.log("The second account ("+web3.eth.accounts[1]+") gifts to the
first one the third content:");
  catalogContract.giftContent(contentsList[2].address, web3.eth.accounts[0],
getParams(web3.eth.accounts[1], contentCost));
  console.log(" - "+contentsList[2].name);
  console.log("Is now tested the purchase of an already purchased content
("+contentsList[2].name+") " +
    "that should raise an error:");
  try {
    grantAccess(contentsList[0].address);
    //catalogContract.getContent(contentsList[2].address,
getParams(web3.eth.accounts[0], value));
  } catch(e) {
    console.log(" - ERROR: "+e.message);
  }
  if (!catalogContract.hasAccess(web3.eth.accounts[0],
contentsList[0].address)
    || !catalogContract.hasAccess(web3.eth.accounts[0],
contentsList[1].address)
    || !catalogContract.hasAccess(web3.eth.accounts[0],
contentsList[2].address))
    throw "The user doesn't have the access right: something went wrong.";
  console.log("Access rights verified: OK.");
  return [contentsList[0], contentsList[1], contentsList[2]];
}

/**
 * Small test of the consumeContent function. Consume a content.
 * @param contentAddress, the consumable content.
 * @param account, the account with which consume the content.
 */
function consumeContent(contentAddress, account = web3.eth.accounts[0]) {
  if (!contentAddress) throw "You must specify the content.";
  return
ContentContract.at(contentAddress).consumeContent(getParams(account));
}

/**
 * Small test about the grantAccess (getContent and giftContent),
 * both the consumeContent functions (Premium and Standard),
 * and the Premium subscription.
 * Produces verbose logs to better understand the behaviour.
 * Are needed at least 3 accounts in web3.eth.accounts and at least 3
contents.
 * @param contentsList, the list of all available contents.
 */
function smallTests(contentsList) {
  console.log("\n\n --- Small tests ---");
  // grant access to web3.eth.accounts[0] on the first 3 contents in
contentsList
  const accessibleContents = grantAccessTest(contentsList);
  // consume the first content and check that is no more consumable
  console.log("\nConsuming the first content: "+accessibleContents[0].name);
  console.log(" - "+accessibleContents[0].address);
  consumeContent(accessibleContents[0].address, web3.eth.accounts[0]);
  if (catalogContract.hasAccess(web3.eth.accounts[0],
accessibleContents[0].address))
    throw "The content still consumable: something went wrong.";
  else console.log("The content is no more consumable: OK.");
  // apply for a premium account
  const premiumCost = catalogContract.premiumCost();
  console.log("\nSubscribing a Premium account on the first account
("+web3.eth.accounts[0]+").");
  catalogContract.buyPremium(getParams(web3.eth.accounts[0], premiumCost));
  console.log(" - isPremium("+web3.eth.accounts[0]+"):
"+catalogContract.isPremium(web3.eth.accounts[0])+
    " (must be true).");
  // gift a premium account
  console.log("\nGifting a Premium account from the first account to the
second one ("+web3.eth.accounts[1]+").");
  catalogContract.giftPremium(web3.eth.accounts[1],
getParams(web3.eth.accounts[0], premiumCost));
  console.log(" - isPremium("+web3.eth.accounts[1]+"):
"+catalogContract.isPremium(web3.eth.accounts[1])+
    " (must be true).");
  // consume the second content and check that it still consumable
  // (should be, because Premium account should not consume previously
bought content)
  console.log("\nConsuming the first content: "+accessibleContents[0].name);
  console.log(" - "+accessibleContents[1].address);
  consumeContent(accessibleContents[1].address, web3.eth.accounts[0]);
  if (!catalogContract.hasAccess(web3.eth.accounts[0],
accessibleContents[1].address))
    throw "The content is no more consumable: something went wrong.";
  else console.log("The content still consumable: OK.");
}

/**
 * Big tests about the statistics functions (getStatistics,
getMostPopularByGenre, getMostPopularByAuthor) and the
 * payout function. Is needed a big amount of contents.
 * @param contentsList, the list of all available contents.
 */
function bigTests(contentsList) {
  console.log("\n\n --- Big tests ---");
  console.log("For each available account, except the last ones, buy and
consume 5 random contents. " +
    "It will take a while.");
  // Exclude the last account for later use
  for (let i = 0; i < web3.eth.accounts.length - 1; i++)
    for (let j = 0; j < 5; j++) {
      const index = rand(contentsList.length - 1);
      // the first account has already bought some contents
      if (!catalogContract.hasAccess(web3.eth.accounts[i],
contentsList[index].address))
        grantAccess(contentsList[index].address, web3.eth.accounts[i]);
      consumeContent(contentsList[index].address, web3.eth.accounts[i]);
    }
  printStatistics(parseStatistics(catalogContract.getStatistics()));

  mostPopularByGenreTest();
  mostPopularByAuthorTest();

  // collectPayout test
  console.log("\n"+web3.toUtf8(latestByAuthor0.name())+" has enough view, so
author "+web3.eth.accounts[0]+" can collect his payout");
  console.log("before - account balance:
"+web3.fromWei(web3.eth.getBalance(web3.eth.accounts[0]))+", " +
    "contract balance: "+web3.eth.getBalance(catalogContract.address));
  catalogContract.collectPayout(getParams());
  console.log("after - account balance:
"+web3.fromWei(web3.eth.getBalance(web3.eth.accounts[0]))+", " +
    "contract balance: "+web3.eth.getBalance(catalogContract.address));
}
```

```
/**
 * Main function of the program.
 * @returns {Promise<void>} because the function is async.
 */
async function main() {
  await connect();
  // deploy the Catalog
  console.log("Deploying catalog...");
  catalogContract = await compileAndDeployContract('CatalogContract.sol');

  // deploy contentsNumber contract from different accounts
  console.log("\nDeploying "+contentsNumber+" contents...");
  const contentContracts = await deployContentsContract(contentsNumber);

  // retrieve contents list
  let contentsList = parseContentsList(catalogContract.getContentsList());
  console.log("\nContents in the catalog:");
  printContentsList(contentsList);

  // check the getNewContentsList
  console.log("\ngetNewContentsList: you should see the last 10 element of
the previous list in the opposite order.");

printContentsList(parseContentsList(catalogContract.getNewContentsList(10)))
;

  // check the suicide function of a content
  console.log("\nTesting the suicide function of a content: we have called
the suicide function on the last item.");
  contentContracts[contentContracts.length - 1]._suicide(
    getParams(contentContracts[contentContracts.length - 1].author()));
  console.log("getNewContentsList: you should see a list very similar to the
preceding one, " +
    "but without the first element.");

printContentsList(parseContentsList(catalogContract.getNewContentsList(10)))
;

  // test getLatestByGenre and getLatestByAuthor functions
  latestByGenreTest();
  latestByAuthorTest();

  // get the new content list after the suicide
  contentsList = parseContentsList(catalogContract.getContentsList());

  // Small test about the grantAccess (getContent and giftContent),
  // both the consumeContent functions (Premium and Standard),
  // and the Premium subscription.
  // Produces verbose logs to better understand the behaviour.
  smallTests(contentsList);

  // Big tests about the statistics functions (getStatistics,
getMostPopularByGenre, getMostPopularByAuthor) and the
  // payout function.
  bigTests(contentsList);

  // check the suicide function of the catalog
  console.log("\nTesting the suicide function of the catalog: all the values
of content contracts should change " +
    "from a value to null. We test it with the name of the first content");
  console.log(" - before: "+ contentContracts[0].name());
  catalogContract._suicide(getParams(catalogContract.owner()));
  console.log(" - after: "+ contentContracts[0].name());
}


main();
```