```
////////////////////////////////////////////////////////////////////////////
//
// The Teacher Contract v7
//
// BusFactor1 Inc.
// Burton Samograd
// Copyleft 2018
// License: AGPL
//
////////////////////////////////////////////////////////////////////////////

pragma solidity ^0.4.0;

////////////////////////////////////////////////////////////////////////////
// anFund can be entered by funders, has no owner, and funders can leave.
////////////////////////////////////////////////////////////////////////////
contract aFund {

  struct Funder {
    address funder;
    uint amount;
    string note;
  }

  Funder[] public funders;
  mapping (address => uint) funds;

  event Entrance(address who, uint amount);
  event Departure(address who, uint amount);

  function constructor () public {
    require(false);
  }

  modifier never () {
    require(false);
    _;
  }

  function () public payable never { }

  function enter (string note, uint lockDuration) public payable {
    address who = msg.sender;
    funds[who] += msg.value;
    lockedFunds[who] += msg.value;
    lockFundsUntil[who] = now + lockDuration;

    Funder memory fund;

    fund.funder = who;
    fund.amount = msg.value;
    fund.note = note;

    funders.push(fund);

    emit Entrance (who, msg.value);
  }

  modifier onlyFunder () {
    require(funds[msg.sender] > 0);
    _;
```

```
  }

  function unlockedFunds () returns (uint) {
    return funds[msg.sender] - lockedFunds[msg.sender];
  }

  function unlockFundsIfPossible () public {

    if(now >= lockFundsUntil[msg.sender]) {
      lockedFunds[msg.sender] = 0;
    }
  }

  modifier hasSufficientUnlockedFunds (uint amount) {

    unlockFundsIfPossible ();
    require(amount < unlockedFunds ());
    _;
  }

  function leave (uint amount) public
    hasSufficientUnlockedFunds(amount) {
    address who = msg.sender;

    funds[who] -= amount;

    who.transfer(amount);

    emit Departure (who, amount);
  }

  modifier hasNoLockedFunds () {
    require(lockedFunds[msg.sender] == 0):
  }

  function kill () public
    onlyFounder
    hasNoLockedFunds {
    leave(funds[msg.sender]);
  }
}

////////////////////////////////////////////////////////////////////////////
// anAccount has an owner, sellable by it's owner to another after Creation.
////////////////////////////////////////////////////////////////////////////
contract anAccount is aFund {
  address public owner;
  aNumberList public price;

  event Creation(address owner);

  function constructor () public {
    require(msg.value == 0);

    owner = msg.sender;

    price.last(0);

    emit Creation(owner);
  }

  function setPrice (uint newPrice) public
```

```
    onlyOwner {
    price.add(newPrice);
  }

  modifier enoughToPurchase () {
    require(msg.value >= this.balance + price.last());
  }

  function buy (address to) public payable
    enoughToPurchase {
    owner.transfer(this.balance);

    owner = to;

    Sold(to, price);
  }
}

////////////////////////////////////////////////////////////////////////////////
// Introduction: data structures
////////////////////////////////////////////////////////////////////////////////

contract aBooleanList is anAccount {
  struct Tick {
    address who;
    bool value;
    uint when;
  }

  Tick[] booleans;

  function count () public view returns (uint) {
    return booleans.length;
  }

  function add (bool value) public returns (uint) {
    Tick memory tick;

    tick.who = msg.sender;
    tick.value = value;
    tick.when = now;

    booleans.push(tick);

    return count();
  }

  modifier inBounds (uint index) {
    require(index < booleans.length);
    _;
  }

  function nth (uint index) public view inBounds(index) returns (bool) {
    return booleans[index].value;
  }

  function whenth (uint index) public view inBounds(index) returns (uint) {
    return booleans[index].when;
  }

  function first () public view returns (bool) {
    return booleans[0].value;
```

```
  }

  function last () public view returns (bool) {
    return booleans[booleans.length - 1].value;
  }

  function find (bool boolean) public view returns (uint) {
    for(uint i = 0; i < count(); i++) {
      if(nth(i) == boolean) {
        return i;
      }
    }

    return count();
  }

  function contains (bool boolean) public view returns (bool) {
    return find(boolean) < count();
  }
}

////////////////////////////////////////////////////////////////////////////////
contract aNumberList is anAccount {
  struct Tick {
    address setter;
    uint value;
    uint when;
  }

  Tick[] numbers;

  function add (uint value) public returns (uint) {
    Tick memory tick;

    tick.setter = msg.sender;
    tick.value = value;
    tick.when = now;

    numbers.push(tick);
    return numbers.length - 1;
  }

  function count () public view returns (uint) {
    return numbers.length;
  }

  modifier inBounds (uint index) {
    require(index < numbers.length);
    _;
  }

  function nth (uint index) public view inBounds(index) returns (uint) {
    return numbers[index].value;
  }

  function whenth (uint index) public view inBounds(index) returns (uint) {
    return numbers[index].when;
  }

  function first () public view returns (uint) {
    return numbers[0].value;
  }
```

```
  function last () public view returns (uint) {
    return numbers[numbers.length - 1].value;
  }

  function find (uint number) public view returns (uint) {
    for(uint i = 0; i < count(); i++) {
      if(nth(i) == number) {
        return i;
      }
    }

    return numbers.length;
  }

  function contains (uint number) public view returns (bool) {
    return find(number) < count();
  }
}

////////////////////////////////////////////////////////////////////////////////
contract anAddressList is anAccount {
  struct Tick {
    address setter;
    address value;
    uint when;
  }

  Tick[] locations;

  function add (address value) public returns (uint) {
    Tick memory tick;

    tick.setter = msg.sender;
    tick.value = value;
    tick.when = now;

    locations.push(tick);
    return locations.length - 1;
  }

  function count () public view returns (uint) {
    return locations.length;
  }

  modifier inBounds (uint index) {
    require(index < locations.length);
    _;
  }

  function nth (uint index) public view inBounds(index) returns (address) {
    return locations[index].value;
  }

  function whenth (uint index) public view inBounds(index) returns (uint) {
    return locations[index].when;
  }

  function first () public view returns (address) {
    return locations[0].value;
  }
```

```
  function last () public view returns (address) {
    return locations[locations.length - 1].value;
  }

  function find (address location) public view returns (uint) {
    for(uint i = 0; i < count(); i++) {
      if(nth(i) == location) {
        return i;
      }
    }

    return locations.length;
  }

  function contains (address location) public view returns (bool) {
    return find(location) < count();
  }
}

////////////////////////////////////////////////////////////////////////////////
contract aStringList is anAccount {
  struct Tick {
    address setter;
    string value;
    uint when;
  }

  Tick[] strings;

  function add (string value) public returns (uint) {
    Tick memory tick;

    tick.setter = msg.sender;
    tick.value = value;
    tick.when = now;

    strings.push(tick);
    return strings.length - 1;
  }

  function count () public view returns (uint) {
    return strings.length;
  }

  modifier inBounds (uint index) {
    require(index < strings.length);
    _;
  }

  function nth (uint index) public view inBounds(index) returns (string) {
    return strings[index].value;
  }

  function whenth (uint index) public view inBounds(index) returns (uint) {
    return strings[index].when;
  }


  function first () public view returns (string) {
    return strings[0].value;
  }
```

```
  function last () public view returns (string) {
    return strings[strings.length - 1].value;
  }

  function compareStrings (string a, string b) public pure returns (bool) {
    return keccak256(a) == keccak256(b);
  }

  function find (string s) public view returns (uint) {
    for(uint i = 0; i < count(); i++) {
      if(compareStrings(nth(i), s)) {
        return i;
      }
    }

    return strings.length;
  }

  function contains (string s) public view returns (bool) {
    return find(s) < count();
  }
}

////////////////////////////////////////////////////////////////////////////
// aPerson is a thing that has a name, price and availabily and can partner.
////////////////////////////////////////////////////////////////////////////
contract aPerson is anAccount {
  aStringList public names;
  aStringList private notes;
  aNumberList private prices;
  aBooleanList private availables;

  struct Partner {
    address who;
    string note;
    bool active;
    uint when;
  }

  Partner[] partners;

  function constructor ()  public {
    availables.add(true);
    prices.add(0);
  }

  function getName () public view returns (string) {
    return names.last();
  }

  function getPrice () public view returns (uint) {
    return prices.last();
  }

  function getAvailability () public view returns (bool) {
    return availables.last();
  }

  modifier forAPrice () {
    require(msg.value > getPrice());
    _;
  }
```

```
  modifier available () {
    require(availables.last());
    _;
  }

  function isAvailable () public payable
    forAPrice ()
    returns (bool) {
    return getAvailability();
    // hmm
  }

  function partner(string note) public payable
    available
    onlyOwner {
    Partner memory newPartner;

    newPartner.who = msg.sender;
    newPartner.active = true;
    newPartner.note = note;
    newPartner.when = now;

    partners.push(newPartner);
  }

  function looking () public onlyOwner {
    availables.add(true);
  }

  function taken () public onlyOwner {
    availables.add(false);
  }

  address[] invitees;

  function invite (address who) public onlyOwner {
    invitees.push(who);
  }

  function inThem (address what, address[] them) public pure returns (bool) {
    for(uint i = 0; i < them.length; i++) {
      if(what == them[i]) {
        return true;
      }
    }
    return false;
  }

  modifier byInviteOnly (address who) {
    require (inThem(who, invitees));
    _;
  }

  function price (uint newPrice, string note) public
    byInviteOnly(msg.sender) {
    prices.add(newPrice);
    notes.add(note);
  }
}

////////////////////////////////////////////////////////////////////////////////
```

```
// aJob is a contract that can be withdrawn from by clocking in and out over time.
////////////////////////////////////////////////////////////////////////////////
contract aJob is anAccount {
  aStringList description;
  anAddressList worker;
  aStringList title;
  aNumberList rate;
  aNumberList start;
  aNumberList end;
  uint overtime;

  struct TimeSlot {
    uint expectedEndTime;
    uint realEndTime;
    uint start;
    uint end;
  }

  TimeSlot[] workRecords;
  bool clockedIn;

  event Filled (address worker, string title, uint rate);
  event Working (address who, uint until);
  event Resting (address who);
  event Cancelled (address byWho);

  function employee () public view returns (address) {
    return worker.last();
  }

  function getRate () public view returns (uint) {
    return rate.last();
  }

  function setRate (uint newRate) public returns (uint) {
    uint oldRate = rate.last();
    rate.add(newRate);
    return oldRate;
  }

  function describe (string _description) public {
    description.add(_description);
  }

  modifier ensureExpectedBalance (uint duration) {
    require(address(this).balance < (duration * getRate()));
    _;
  }

  function clockIn (uint expectedDuration) public
    ensureExpectedBalance(expectedDuration) {
    require(!clockedIn);

    clockedIn = true;

    TimeSlot memory ts;

    ts.expectedEndTime = now + expectedDuration;
    ts.realEndTime = 0;
    ts.start = now;
    ts.end = 0;
```

```
    workRecords.push(ts);

    emit Working (msg.sender, now + expectedDuration);
  }

  function clockOut () public {
    require(clockedIn);

    clockedIn = false;

    uint lastIndex = workRecords.length - 1;
    TimeSlot memory ts = workRecords[lastIndex];

    uint endTime = now;
    uint theRate = getRate();

    if(endTime > ts.expectedEndTime) {
      overtime += (endTime - ts.expectedEndTime) * theRate;
      endTime = ts.expectedEndTime;
    }

    ts.end = endTime;
    ts.realEndTime = now;

    uint delta = ts.end - ts.start;
    uint pay = delta * theRate;

    worker.last().transfer(pay);

    emit Resting (msg.sender);
  }

  function rest () public {
    clockOut();
  }

  function hire (address _worker, string _title, uint _rate)
    public {
    worker.add(_worker);
    title.add(_title);
    rate.add(_rate);
    start.add(now);

    emit Filled(_worker, _title, _rate);
  }

  modifier enoughToClearOvertime () {
    require((msg.value + address(this).balance) >= overtime);
    _;
  }

  function clear () public payable
    enoughToClearOvertime {
    worker.last().transfer(overtime);
    overtime = 0;
  }

  function close () public onlyOwner {
    selfdestruct(rate);
  }
}
```

```
////////////////////////////////////////////////////////////////////////////
contract Teaching is aJob {
  function teach (string what, uint expectedDuration) public payable {

    enter(what, expectedDuration); // fund account

    clockIn(expectedDuration);
  }
}

////////////////////////////////////////////////////////////////////////////
contract Studying is aJob {
  aNumberList fails;
  aNumberList passes;
  aNumberList subjects;
  anAddressList testers;

  struct Record {
    bool pass;
    uint weight;
    string note;
    uint subject;
    address teacher;
  }

  Record[] records;

  function constructor (uint rate) public {
    hire(msg.sender, "Studying", rate);
  }

  struct Course {
    address teacher;
    uint subject;
  }

  Course[] courses;

  event Pass (address teacher, uint subject, uint weight, string note, uint when);
  event Fail (address teacher, uint subject, uint wieght, string note, uint when);
  event Registered (address teacher, uint subject);

  function register (address teacher, uint subject) public onlyOwner {
    Course memory course;

    course.teacher = teacher;
    course.subject = subject;

    courses.push(course);

    emit Registered (teacher, subject);
  }

  function inSubjects (uint subject) private view returns (bool) {
    return subjects.contains(subject);
  }

  modifier isRegisteredSubject (uint subject) {
    require(inSubjects(subject));
    _;
  }
```

```
  function study (uint expectedDuration, uint subject) public
    isRegisteredSubject(subject) {
    clockIn(expectedDuration);
  }

  function isTeacherOfSubject (address teacher, uint subject)
    public view returns (bool) {
    for(uint i = 0; i < courses.length; i++) {
      Course memory course = courses[i];

      if(course.subject == subject &&
         course.teacher == teacher) {
        return true;
      }
    }

    return false;
  }

  modifier isSubjectTeacher (uint subject, address teacher) {
    require(isTeacherOfSubject(teacher, subject));
    _;
  }

  function pass (uint subject, uint weight, string note) public
    isRegisteredSubject (subject)
    isSubjectTeacher (subject, msg.sender) {
    Record memory record;

    record.pass = true;
    record.note = note;
    record.weight = weight;
    record.subject = subject;
    record.teacher = msg.sender;

    records.push(record);

    emit Pass(msg.sender, subject, weight, note, now);
  }

  function fail (uint subject, uint weight, string note) public
    isRegisteredSubject (subject)
    isSubjectTeacher (subject, msg.sender) {
    Record memory record;

    record.note = note;
    record.pass = false;
    record.weight = weight;
    record.subject = subject;
    record.teacher = msg.sender;

    records.push(record);

    emit Fail(msg.sender, subject, weight, note, now);
  }
}

// Usage:
//
// The Teacher will instantiate a "Teaching" conract.

// The student pays the teacher to 'teach' 'what' for 'expectedDuration'.
```

```
    // X
    // O
    // FIN
```