

The purpose of this homework is:

- to practice reading and understanding a large piece of software
- to practice developing and maintaining mutable structures
- to practice using Java iterators
- to have some fun

You are allowed to work in pairs on this homework, although it is not required for you to do so. You are responsible for keeping your code hidden from all other students. Part of your grade will be determined by how well you hide your code, part of your grade will be determined by how well you follow the instructions for submitting your code, part of your grade will depend upon the correctness of your code, and part of your grade will depend upon the readability of your code (e.g. formatting and comments).

By 22h00 on the due date, using the Web-based submission system at

<https://cgi.ccs.neu.edu/home/vslav/csu370/csu370.php>

you should submit source files containing your solution to the assignment. Each file you submit should begin with a block comment that lists:

- (1) Your names, as you want the instructor to write them.
- (2) Your email addresses.
- (3) Any remarks that you wish to make to the instructor.

Specific details of which files to submit will be given below.

Late assignments may be discounted, and very late assignments may be discarded.

The CSU 370 Adventure Game

In the mid-70s, a very interesting crop of games emerged that have been variously called adventure games, and more recently interactive fiction games. The idea of those games is to provide a simulation of some world, populated with objects and characters with which a player could interact by way of typing at the keyboard short mnemonic commands such as **TAKE APPLE** and **FIRE SHOTGUN AT BOBBY**.

It turns out that writing such simulation code is pretty much what object-oriented programming was invented for in the 60s. Indeed, the first object-oriented language was called Simula, and was a language for writing simulations.

In this homework, inspired by and liberally adapted from a long running series of homeworks from MIT's 6.001 course on structure and interpretation of computer programs, you will be studying, understanding, and modifying such an adventure game.

The code is available on the course web page, and there is a lot of it. Your first task is to get an understanding for how the code is structured, which classes are in the system, and how they interact. We will help you in

this endeavour by describing the overall structure and the core classes of the game. But you will have to do some of the leg work and look at the code to fill in the blanks in this description.

This homework looks like it is long, but most of it is simply describing the code. The main point of the homework is really to get you to sit down and understand a software system, which is a difficult skill but one that you must learn at some point.

After you have understood the code, once you have a good idea of how things are put together and why they work the way they do, **only then** should you start working on the exercises listed at the end of this homework.

Classes for a Simulated World

The objects that inhabitate the simulated world we will call *artifacts* because the term objects is too easily confused with Java objects. Let us take a brief tour of some of the most important of those classes.

Artifact Class

The `Artifact` class is the class from which all artifacts subclass. It implements the basic functionality that every artifact should manifest.

It is small enough that we can give it here.

```
public class Artifact {

    private String name;

    protected Artifact (String n) {
        this.name = World.noSpaces(n);
    }

    public static Artifact create (String name) {
        Artifact artifact = new Artifact(name);
        artifact.install();
        return artifact;
    }

    public String name () {
        return this.name;
    }

    public void install () { }

    public void destroy () { }
}
```

The shape of this class informs that shape of all other artifact classes. The constructor is protected, so that it is available from possible inheriting subclasses. This also means that to create an object of the `Artifact` class, we need to invoke the static creator method `Artifact.create`, passing in the name of the artifact as a string. Note that this string is first converted so that it contains no spaces (by converting every space into a -), and so that it is in lowercases. The reason for this is that the parser that will read user input (see

below) will automatically split the input at spaces, and therefore we will not be able to refer to any artifact that has a name containing spaces. We can query an artifact for its name by invoking the `name` method.

After being created, every artifact will invoke an `install` method that installs the artifact in the world. By default, the method does nothing. (But see below in the `Thing` class.) Similarly, every artifact has a `destroy` method that removes the artifact from the world. Again, by default, the method does nothing.

Thing Class

A `Thing` is probably the simplest class that implements a useful artifact. It is meant to represent an artifact that has a location in the world. A location is represented by an artifact implementing the `Container` interface. (There are two such basic artifacts, `Person` and `Place`, which we will return to later.)

The `Thing` class extends the `Artifact` class, meaning that in particular every `Thing` has a name.

```
public class Thing extends Artifact {

    private Container location;

    protected Thing (String n, Container l) {
        super(n);
        location = l;
    }

    public static Thing create (String n, Container l) {
        Thing obj = new Thing(n,l);
        obj.install();
        return obj;
    }

    public void install () {
        super.install();
        this.location().addThing(this);
    }

    public Container location () {
        return this.location;
    }

    public void destroy () {
        this.location().delThing(this);
        super.destroy();
    }

    public void emit (String text) {
        World.tellRoom (this.location(),
            "At " + this.location().name() + " " + text);
    }

    ...

}
```

Note that the constructor for the class invokes the `Artifact` constructor. Creating a `Thing` requires passing in the name of the artifact, as well as a location in which to create it. Upon installation, the newly created `Thing` adds itself to the list of artifacts contained at the desired location. Symmetrically, when destroyed, a `Thing` will remove itself for the list of artifacts contained at its location. The method `emit` is used to write a message on the screen that is only visible when the player is in the same room as the artifact that emits the message. (It relies on the static method `World.tellRoom`. Many useful auxiliary functions are defined as static methods of the `World` class, described later.)

Place Class

Our simulated world needs places (e.g., rooms) where interesting things can happen. A `Place` is an artifact that extends the `Artifact` class.

```
public class Place extends Artifact implements Container {

    private Container content;
    private LinkedList<Exit> exits;

    protected Place (String n) {
        super(n);
        this.content = ContainerProxy.make();
        this.exits = new LinkedList<Exit>();
    }

    public static Place create (String n) {
        Place obj = new Place(n);
        obj.install();
        return obj;
    }

    public LinkedList<Exit> exits () {
        return this.exits;
    }

    private Exit findExitInDirection(String direction) {
        for (Exit exit : this.exits)
            if (direction.equals(exit.direction()))
                return exit;
        return null;
    }

    public Exit exitTowards (String direction) {
        return this.findExitInDirection(direction);
    }

    public void addExit (Exit exit) {
        String direction = exit.direction();
        if (this.exitTowards(direction)!=null)
            throw new RuntimeException (this.name() + " already has exit " + direction);
        else
            this.exits.add (exit);
    }
}
```

```

    }

    public LinkedList<Thing> things () {
        return content.things();
    }

    public boolean haveThing (Thing t) {
        return content.haveThing(t);
    }

    public void addThing (Thing t) {
        content.addThing(t);
    }

    public void delThing (Thing t) {
        content.delThing(t);
    }

    public Exit randomExit () {
        if (exits.isEmpty()) return null;
        int index = World.random(exits.size());
        return exits.get(index);
    }
}

```

Because a `Place` has a lot of functionality, it is somewhat complicated. First, a `Place` has some content. It implements the `Container` interface given by:

```

public interface Container {
    public String name ();
    public LinkedList<Thing> things ();
    public boolean haveThing (Thing t);
    public void addThing (Thing t);
    public void delThing (Thing t);
}

```

The container interface is really just a thin layer over an implementation of list of `Things`

We represent the content of a `Place` by an object `ContainerProxy` that also implements the `Container` interface, and that simply defines a linked list of `Things` (using the `LinkedList<E>` class in the Java Collections Framework). The methods in `Place` that interact with the content simply delegate those methods to that `content` object. (We will let you look at the code of `ContainerProxy` by yourself. The most important thing to note when you look at the code is the method `things`, which returns the list of contained things, that creates a *new* list to hold the content. This is our attempt at controlling the mayhem caused by using mutable structures.)

When a `Place` is created, the content is initialized to be empty. If you recall the `Thing` class above, you will remember that when a `Thing` is installed, it adds itself to the content of a location; if that location is a `Place`, it will add itself to the content of the `Place` via the `addThing` method of the `Place`.

The remainder of the complexity in the `Place` class has to do with exits. A `Place` has exits in at most six possible directions, `north`, `south`, `east`, `west`, `up`, and `down` (although other directions are possible, although the game will not understand them). An exit is an artifact of class `Exit` (which, because it extends

Artifact, has a name, which is the direction of the exit), and connects a **Place** to another. We will let you examine the code for **Exit**, and understand how exits are managed by **Place**. (Note that a place is initially created without exits; exits are added using the **addExit** method of a **Place**.)

MobileThing Class

Now that we have things that can be contained in some place, it is natural to have things that can move and change location. A **MobileThing** is an artifact that extends **Thing** and that implements exactly that functionality.

Here is a sketch of the class, where some uninteresting bits elided. (As usual, see the code for the exact content.)

```
public class MobileThing extends Thing {

    private Container mobileLocation;

    protected MobileThing (String n, Container l) {
        super(n,l);
        this.mobileLocation = l;
    }

    ...

    public Container location () {
        return this.mobileLocation;
    }

    public void changeLocation (Container l) {
        this.mobileLocation.delThing(this);
        l.addThing(this);
        this.mobileLocation = l;
    }

    public void destroy () {
        super.destroy();
        // move the destroyed object to Limbo
        this.mobileLocation=Limbo.getInstance();
    }

    ...

    public Container creationSite () {
        return super.location ();
    }

    ...
}
```

What's important is that a **MobileThing** has a private location field that holds the current (mobile) location of the artifact. Because it extends a **Thing** that also holds a location, that last location can be used to get

back the original creation site of the artifact. (See the `creationSite` method.) A `MobileThing` implements a `changeLocation` method that can move the object from one location to the another, where a location is an artifact that implements the `Container` interface. Note also that when a `MobileThing` is destroyed, it is moved to `Limbo`, a special place defined in the code, with no exits.

Person Class

A `Person` is a kind of mobile thing that is also a container: `Persons` can carry things around, and serve as locations. The definition of a person is more complex than that of a place, and here are important excerpts:

```
public class Person extends MobileThing implements Container {

    private static int initialHealth = 3;

    private Container content;
    private int health;

    protected Person (String name, Place birthplace) {
        super(name,birthplace);
        this.content = ContainerProxy.make();
        this.health = initialHealth;
    }

    public static Person create (String name, Place birthplace) { ... }

    public int health () {
        return this.health;
    }

    public void say (String arg) {
        World.tellRoom ((Place) this.location(),
            "At " + this.location().name() + " " + this.name() +
            " says -- " + arg);
    }

    public LinkedList<Person> peopleAround () { ... }
    public LinkedList<Thing> stuffAround () { ... }
    public LinkedList<Thing> peekAround () { ... }

    public Thing take (Thing t) { ... }
    public void drop (MobileThing t) { ... }

    public boolean go (String direction) { ... }

    public void suffer (int hits) { ... }

    public void die () { ... }

    ...
}
```

A **Person** gets created like a **MobileThing** with a name and an initial location, and it gets assigned some empty content and an initial health value. Health decreases when the **Person** is made to **suffer** (e.g., by being bit by a **Troll**), and when health goes down below 0, the **Person** dies.

Useful methods for a **Person** include **say**, which prints a message on the screen stating that the **Person** says something, visible only if the player is in the same room as the **Person** saying something. A **Person** can **take** an artifact (which must be a **MobileThing**), after which the taken artifact is moved from its current location to the content of the **Person**, and a carried artifact can be **dropped**. A **Person** can be made to **go** in a direction.

Other important methods include **peopleAround** that returns a list (as an instance of **LinkedList<Person>**) of other **Persons** in the same location, **stuffAround** that returns a list of **Things** in the same location, and **peekAround** that returns a list of all **Things** carried by **Persons** in the same location. Please see the code for the details of these (and other) methods.

Avatar Class

An important subclass of **Person** is **Avatar**, which is used to represent the player. Interaction in the game essentially amounts to you issuing directives to your avatar.

Some of the methods inherited from **Person** are overwritten to provide for specific functionality; for example, when the avatar dies, the game is over.

```
public class Avatar extends Person {  
  
    protected Avatar (String name, Place birthplace) {  
        super(name,birthplace);  
    }  
  
    public static Avatar create (String name, Place birthplace) { ... }  
  
    public void lookAround () { ... }  
  
    ...  
}
```

Method **lookAround** is interesting and worth looking at in the code. It is used to print a friendly message during the message describing the content of the room you are in: other people, other things, what you are carrying. If it is invoked automatically whenever you successfully change locations, or whenever you ask for it explicitly.

AutonomousPerson Class

What makes a simulated world interesting is activity that is not controlled by the player. In particular, we would like to have **Persons** moving about, and interacting with each others and the player. To implement those, we introduce an **AutonomousPerson** artifact that extends **Person**, and is worth describing in some detail.

```
public class AutonomousPerson extends Person {  
  
    private int restlessness;  
    private int miserly;
```

```

private int actRecord;

protected AutonomousPerson (String name, Place birthplace,
                             int restlessness, int miserly) {
    super(name,birthplace);
    this.restlessness = restlessness;
    this.miserly = miserly;
}

public static AutonomousPerson create (String name, Place birthplace,
                                       int r, int m) { ... }

public void install () {
    super.install();
    this.actRecord = Clock.registerAction(new MoveAndTakeStuffAction(this));
}

private static class MoveAndTakeStuffAction implements Action {
    private AutonomousPerson person;
    public MoveAndTakeStuffAction (AutonomousPerson p) {
        this.person = p;
    }
    public void perform () {
        this.person.moveAndTakeStuff();
    }
}

public void moveAndTakeStuff () {
    if (World.random(this.restlessness) == 0)
        this.moveSomewhere();
    if (World.random(this.miserly) == 0)
        this.takeSomething();
}

public void die () {
    Clock.removeAction(this.actRecord);
    this.say("SHREEEEEEK! I, uh, suddenly feel very faint...");
    super.die();
}

public void moveSomewhere () {
    Exit exit = ((Place) this.location()).randomExit();
    if (exit!=null)
        this.goExit(exit);
}

public void takeSomething () {
    LinkedList<Thing> pickFrom = this.stuffAround();
    pickFrom.addAll(this.peekAround());
}

```

```

    if (!pickFrom.isEmpty())
        this.take(World.pickRandom(pickFrom));
}

...
}

```

An `AutonomousPerson` is created with a name, an initial location, a restlessness factor (indicating the likelihood that the person tries to move at every turn), and a miserly factor (indicating the likelihood that the person tries to take something from its location at every turn).

How do we tell the `AutonomousPerson` to try to do something at every turn? We do so by using a clock to keep track of turns, and using *actions* to make the clock invoke specific methods of the `AutonomousPerson` at every turn. Let's go over this slowly.

First off, the clock is implemented by a class `Clock` with the following signature:

```

public class Clock {

    public static int time();
    public static void tick ();
    public static int registerAction (Action act);
    public static void removeAction (int id);

}

```

Note that all the methods here are static, so that the class essentially acts as a singleton class. (We could have used the Singleton Pattern, but since we never need to pass the clock around, we elected not to do so.) Method `time` returns the current time, that is, the number of turns elapsed since the beginning of the game, where a turn ends when the player either changes location, or waits the turn out. (See below the interaction section.) Method `tick` makes the clock tick once, that is, increments the turn counter, and invokes all the actions that have been registered with the clock.

To register an action, we invoke method `registerAction`, passing in an instance of an action. An action is a class implementing the following simple interface:

```

public interface Action {
    public void perform ();
}

```

Invoking an action really is just invoking the method `perform` in the action. Thus, to execute a given method `f` at every turn, we simply need to wrap it up in a class implementing the `Action` interface, have the `perform` method of that class invoke our `f` method, and register the action with the clock. Method `install` in `AutonomousPerson` does just that, registers an action that invokes the method `moveAndTakeStuff` of the autonomous person. See the code for the exact details of how this is done, and make sure you understand what is going on.

Registering an action returns a unique identifier (as an integer) that can be used to later on remove the action from the clock, by invoking `Clock.removeAction`, passing in the identifier of the action to remove. Removing actions is usually done when an artifact is destroyed, so that the clock does not attempt to invoke methods on that artifact anymore. (See the `die` method of `AutonomousPerson`, for instance.)

The `moveAndTakeStuff` method that is invoked by the action uses random numbers as implemented through a utility method in `World` to determine whether to move somewhere (method `moveSomewhere`) and whether to take something (method `takeSomething`).

The Interactive Loop

Look at the `Adventure` class, which implements the `main` method of the game. When it is called, it does a couple of things. The most important of those are calling `createWorld`, which initializes the world by creating all the artifacts, including all the places making up the simulated world, all the things populating the simulate world, and all the persons living in the simulated world. A very important person it creates is the avatar, which represents you, the player. In fact, the argument to the `createWorld` method is the name by which you will be known in the simulated world. You can access the avatar artifact through the special `World.me` method, which always returns the avatar artifact in the world. After creating the world, the `main` method creates the vocabulary, which tells the system how to interpret input you will give to the interactive interface into actions to be performed on the artifacts of the simulated world. After these acts of creation, the `main` method yields control to the `Interaction.mainLoop` method, which implements the interactive interface. More on that in a second.

Look at the `createWorld` method. As we said, it creates all the artifacts populating the world. For many of those artifacts, their initial location is randomly picked, by picking a room at random. The method `World.rooms` returns all the rooms that have been registered with the world. (This is what you want to call `World.registerRoom` when creating a room.)

Some of the code in `createWorld` is commented out, because it will not work until you complete the exercises below.

The `Interaction` class implements the interactive loop. It is in charge of querying an input from the player, in the form of a sequence of words such as:

```
? north
```

or

```
? take unfinished-hw-4
```

or

```
? give graded-hw-4 riccardo
```

The first word of the input is the action to be performed (where here we consider directions to be actions, namely the action of going in the specified direction) by the avatar. If there is a second word, it is the direct object of the action. If there is a third word, it is the indirect object of the action.

The interactive loop, after parsing the input and recognizing the words, will attempt to find a verb in its vocabulary that corresponds to the action, by calling `World.findVerb`, which looks through the list of registered verbs. (A verb is registered by calling, naturally enough, `World.registerVerb`—see method `createVocabulary` in `Adventure`.) After having identified a verb, it attempts to find an artifact in the location of the avatar that has the same name as the direct object if one was specified, and also an artifact that has the same name as the indirect object if one was specified, in both cases by calling the method `World.thingNamed`.

Once the verb, the direct object, and indirect object (if needed) have been identified, then the `action` method of the verb is invoked, and passed the direct and indirect object (if needed). That `action` method describes how to affect the simulated world. For instance, consider the `Take` verb:

```
public class Take extends Verb {  
  
    protected Take () {
```

```

    super("take");
}

public static Verb create () {
    return new Take();
}

public Result action (Thing obj) {
    World.me().take(obj);
    return Result.STOP;
}
}

```

The constructor for `Take` calls the `Verb` constructor, passing in the word representing the verb during parsing. The `action` method takes a direct object, and invokes the `take` method of the avatar to take the specified object. All actions return either `Result.STOP` or `Result.CONTINUE`; if the former, then performing the action does not affect the clock, while in the latter case, after the action is performed, the clock is move forward one tick. (Only movement actions and waiting actions move the clock forward.)

There are quite a few verbs that have been supplied. Here is a short description of them.

<code>quit</code>	stop the game
<code>debug</code>	toggle debugging mode (lets you see all comments even in rooms you are not in)
<code>look</code>	describes what is around you
<code>wait</code>	move clock forward one tick
<code>take a</code>	take artifact <code>a</code> (must be a mobile thing)
<code>drop a</code>	drop artifact <code>a</code> (must be carried)
<code>give a p</code>	gives artifact <code>a</code> (must be carried) to <code>p</code> (a person)
<code>use a</code>	use artifact <code>a</code> (must implement <code>Usable</code> interface)
<code>ask p</code>	ask person <code>p</code> something (what depends on the person)
<code>north</code>	go north
<code>south</code>	go south
<code>east</code>	go east
<code>west</code>	go west
<code>up</code>	go up
<code>down</code>	go down

The goal of the game, as it stands, is to pass the course. To do so, you need to `ask` a professor when that professor has a graded homework in his possession, and he or she will pass you. (This is easier than in real life, we expect.) The easiest way to achieve that, of course, is to simply `give` a graded homework to a professor.

To obtain a graded homework, you need to `ask` a grader when that grader has a completed homework in his or her possession, and she will then drop a graded homework on the floor. (In fact, he or she will drop a graded homework corresponding to every completed homework he or she has in her possession.)

And how do you obtain a completed homework? Well, you first have to find an unfinished homework somewhere (there are a few around), take it, go to one of the two computer labs (they are both in West Village H), and `use` the computer you find there. Using a computer while possession an unfinished homework will consume that homework and produce a completed homework in your location.

Sounds easy? Beware of trolls that are out to eat you, and of PRL graduate students who are out to burn all non-scheme programming assignments.

To give you a sense of what is going, here is a sample interaction with the game. Note that this interaction assumes that the exercises below have been done.

The CSU 370 Adventure Game, version 1.0 (November 2007)

You are in lake-hall
You are holding: tr3000
There is no stuff in the room
You see other people: sophie-sophomore
The exits are in directions: west south east

What is thy bidding? west
blubbering-fool moves from lake-hall to knowles-center
The clock ticks 1
At knowles-center sophie-sophomore says -- Hi blubbering-fool

You are in knowles-center
You are holding: tr3000
There is no stuff in the room
You see other people: sophie-sophomore
The exits are in directions: south east north west

What is thy bidding? south
blubbering-fool moves from knowles-center to wvh-first-floor
At wvh-first-floor blubbering-fool says -- Hi riccardo
The clock ticks 2
At wvh-first-floor vlad says -- Hi riccardo blubbering-fool
riccardo moves from wvh-first-floor to knowles-center

You are in wvh-first-floor
You are holding: tr3000
There is no stuff in the room
You see other people: vlad
The exits are in directions: up south west north east

What is thy bidding? west

/// cut many moves to actually find a homework

blubbering-fool moves from speare-hall to marino-center
The clock ticks 14

You are in marino-center
You are holding: tr3000
You see stuff in the room: unfinished-hw-6
There are no other people around you
The exits are in directions: east south north

What is thy bidding? take unfinished-hw-6
At marino-center blubbering-fool says -- I take unfinished-hw-6 from marino-center

What is thy bidding? east
blubbering-fool moves from marino-center to knowles-center
The clock ticks 15

You are in knowles-center
You are holding: tr3000 unfinished-hw-6
There is no stuff in the room
There are no other people around you
The exits are in directions: south east north west

What is thy bidding? south
blubbering-fool moves from knowles-center to wvh-first-floor
The clock ticks 16
At wvh-first-floor vlad says -- Hi blubbering-fool
At wvh-first-floor vlad says -- I take unfinished-hw-6 from blubbering-fool
At wvh-first-floor blubbering-fool says -- I lose unfinished-hw-6
At wvh-first-floor blubbering-fool says -- Yaaaaah! I am upset!

You are in wvh-first-floor
You are holding: tr3000
There is no stuff in the room
You see other people: vlad
The exits are in directions: up south west north east

What is thy bidding? take unfinished-hw-6
At wvh-first-floor blubbering-fool says -- I take unfinished-hw-6 from vlad
At wvh-first-floor vlad says -- I lose unfinished-hw-6
At wvh-first-floor vlad says -- Yaaaaah! I am upset!

What is thy bidding? west
blubbering-fool moves from wvh-first-floor to wvh-computer-lab
The clock ticks 17

You are in wvh-computer-lab
You are holding: tr3000 unfinished-hw-6
You see stuff in the room: hal-9000
There are no other people around you
The exits are in directions: east

What is thy bidding? use hal-9000
At wvh-computer-lab blubbering-fool says -- Okay, time to finish unfinished-hw-6

What is thy bidding? take completed-hw-6
At wvh-computer-lab blubbering-fool says -- I take completed-hw-6 from wvh-computer-lab

/// good, now just gotta find a grader - let's use our handy dandy gps-tracker...

What is thy bidding? use tr3000
At wvh-computer-lab blubbering-fool says -- I fiddle with the buttons on tr3000
At wvh-computer-lab blubbering-fool says -- blubbering-fool is in wvh-computer-lab
At wvh-computer-lab blubbering-fool says -- olin is in cs-office

At wvh-computer-lab blubbering-fool says -- riccardo is in curry-center
At wvh-computer-lab blubbering-fool says -- sam is in snell-library
At wvh-computer-lab blubbering-fool says -- sophie-sophomore is in snell-library
At wvh-computer-lab blubbering-fool says -- cedric-senior is in snell-library
At wvh-computer-lab blubbering-fool says -- vlad is in knowles-center

/// ah, vlad is close by

What is thy bidding? east
blubbering-fool moves from wvh-computer-lab to wvh-first-floor
The clock ticks 18

You are in wvh-first-floor
You are holding: tr3000 completed-hw-6
There is no stuff in the room
There are no other people around you
The exits are in directions: up south west north east

What is thy bidding? north
blubbering-fool moves from wvh-first-floor to knowles-center
The clock ticks 19
At knowles-center sophie-sophomore says -- Hi blubbering-fool
At knowles-center sophie-sophomore says -- I take tr3000 from blubbering-fool
At knowles-center blubbering-fool says -- I lose tr3000
At knowles-center blubbering-fool says -- Yaaaaah! I am upset!

You are in knowles-center
You are holding: completed-hw-6
There is no stuff in the room
You see other people: sophie-sophomore
The exits are in directions: south east north west

What is thy bidding? take tr3000
At knowles-center blubbering-fool says -- I take tr3000 from sophie-sophomore
At knowles-center sophie-sophomore says -- I lose tr3000
At knowles-center sophie-sophomore says -- Yaaaaah! I am upset!

/// he moved, where is he?

What is thy bidding? use tr3000
At knowles-center blubbering-fool says -- I fiddle with the buttons on tr3000
At knowles-center blubbering-fool says -- olin is in wvh-second-floor
At knowles-center blubbering-fool says -- riccardo is in curry-center
At knowles-center blubbering-fool says -- vlad is in au-bon-pain
At knowles-center blubbering-fool says -- sam is in snell-library
At knowles-center blubbering-fool says -- cedric-senior is in snell-library
At knowles-center blubbering-fool says -- blubbering-fool is in knowles-center
At knowles-center blubbering-fool says -- sophie-sophomore is in knowles-center

What is thy bidding? west
blubbering-fool moves from knowles-center to marino-center

The clock ticks 20
At marino-center vlad says -- Hi blubbering-fool

/// gotcha!

You are in marino-center
You are holding: completed-hw-6 tr3000
There is no stuff in the room
You see other people: vlad
The exits are in directions: east south north

What is thy bidding? give completed-hw-6 vlad
At marino-center blubbering-fool says -- I drop completed-hw-6 at marino-center
At marino-center vlad says -- I take completed-hw-6 from marino-center

What is thy bidding? ask vlad
At marino-center vlad says -- Let's see. What's this? An infinite loop? Mmmm...
At marino-center vlad finishes grading completed-hw-6

What is thy bidding? take graded-hw-6
At marino-center blubbering-fool says -- I take graded-hw-6 from marino-center

/// okay, not time to find a professor and hand this in...

/// cut moves to find riccardo

blubbering-fool moves from curry-center to ell-hall
At ell-hall blubbering-fool says -- Hi riccardo
The clock ticks 25
At ell-hall riccardo says -- I take graded-hw-6 from blubbering-fool
At ell-hall blubbering-fool says -- I lose graded-hw-6
At ell-hall blubbering-fool says -- Yaaaaah! I am upset!

/// cute, the prof stole my homework... no worries, I would have given it to him anyways

You are in ell-hall
You are not holding anything
There is no stuff in the room
You see other people: riccardo
The exits are in directions: east west

What is thy bidding? ask riccardo
At ell-hall riccardo says -- Look at that, someone at least will pass this course!
At ell-hall riccardo says -- Congratulations!

If you use debug mode (by using the `debug` verb), then you get to see everything that is happening in the world, which is quite useful at times.

The CSU 370 Adventure Game, version 1.0 (November 2007)

You are in olin-lair

You are holding: tr3000
You see stuff in the room: htdp
You see other people: cedric-senior
The exits are in directions: north

What is thy bidding? debug
Setting debug mode to true

What is thy bidding? wait
The clock ticks 1
sophie-sophomore moves from lake-hall to willis-hall
olin moves from prl-lab to wvh-third-floor
At wvh-third-floor olin's belly rumbles

You are in olin-lair
You are holding: tr3000
You see stuff in the room: htdp
You see other people: cedric-senior
The exits are in directions: north

What is thy bidding? wait
The clock ticks 2
At snell-library joe-junior says -- I take cs-book from snell-library
cedric-senior moves from olin-lair to prl-lab
riccardo moves from ell-hall to curry-center
sam moves from wvh-108 to wvh-first-floor
At wvh-second-floor polyphemus's belly rumbles
At wvh-third-floor olin's belly rumbles

You are in olin-lair
You are holding: tr3000
You see stuff in the room: htdp
There are no other people around you
The exits are in directions: north

What is thy bidding? wait
The clock ticks 3
jeff moves from marino-center to au-bon-pain
At knowles-center vlad says -- I take unfinished-hw-5 from knowles-center
sam moves from wvh-first-floor to wvh-108
polyphemus moves from wvh-second-floor to wvh-first-floor
At wvh-third-floor olin's belly rumbles

You are in olin-lair
You are holding: tr3000
You see stuff in the room: htdp
There are no other people around you
The exits are in directions: north

What is thy bidding? wait
The clock ticks 4

jeff moves from au-bon-pain to marino-center
At curry-center riccardo says -- I take unfinished-hw-2 from curry-center
At wvh-108 sam says -- I take unfinished-hw-6 from wvh-108
At wvh-108 sam says -- Wait a minute! This is not Scheme...
At wvh-108 sam says -- Burn, baby, burn!
At wvh-108 sam burns unfinished-hw-6
At sam unfinished-hw-6 is destroyed
polyphemus moves from wvh-first-floor to wvg
At wvh-third-floor olin's belly rumbles

You are in olin-lair
You are holding: tr3000
You see stuff in the room: htdp
There are no other people around you
The exits are in directions: north

What is thy bidding? wait
The clock ticks 5
riccardo moves from curry-center to knowles-center
At knowles-center riccardo says -- Hi vlad
At wvh-third-floor olin's belly rumbles

Exercise 0: Stretch Your Coding Muscles (4 pts)

This first exercise is just a warm-up. I just want you to add to the world somewhat.

- Add two new **Places** to the world, and connect them to existing rooms.
- Add five new **Things** (some mobile, some not).

To do this, modify the method `Adventure.createWorld`.

What to submit for this exercise: your modified file `Adventure.java`.

Exercise 1: Implement a GPS Tracker (7 pts)

Because to win the game you have to be able to find both a grader and a professor, it makes sense to have a way to find people in the simulated world. To do so, implement a new class for a `GPSTracker` artifact. Feel free to use other classes in the game as a guide. (For the strange `isA` method and the corresponding nested static class, see Exercise 2 below.) Give a thought as to what class you want it to extend. What will you want to be able to do with a `GPSTracker`?

Because we want to be able to use such a tracker, the class needs to implement the `Usable` interface. (See the `Computer` class to see an example of another class implementing the `Usable` interface.) This means, in particular, that your class should implement a method `use` that takes a single argument (the `Person` using the GPS tracker).

This `use` method needs to output all the `Persons` in the simulated world, with their location. You need to figure out a way to do that using the methods provided. You can use the sample output above as a guide for how to report the information, but feel free to use a different output format.

Create an instance of a `GPSTracker` in `Adventure.createWorld` to test out your code.

What to submit for this exercise: your file `GPSTracker.java`

Exercise 2: Implement the hasA Method (7 pts)

The `Person` class has an incomplete method

```
public LinkedList<Thing> hasA (Predicate pred) {  
  
    /* Write me for Exercise 2  
     * replacing the next line by the actual code  
     */  
    return new LinkedList<Thing>();  
}
```

Your job here is to complete it. Here is what the method does. It takes as input an object that implements `Predicate`, which is an interface defined as follows:

```
public interface Predicate {  
    public boolean check (Thing obj);  
}
```

Intuitively, an object `pred` implementing `Predicate` is a *predicate* on `Things`. Given a `Thing t`, invoking

```
pred.check(t)
```

returns true or false depending on whether `t` satisfies the predicate. To make up a silly example, the following predicate checks whether a `Thing` has name `billy-bob`:

```
class BillyBobPred implements Predicate {
    public BillyBobPred () {}
    public boolean check (Thing obj) {
        return (obj.name().equals("billy-bob"));
    }
}
```

Thus, we can pass an object `new BillyBobPred()` to the `hasA` method.

What does the method do anyways? It returns the list of all `Things` that the person possesses (recall that `hasA` is a method in class `Person`) for which the predicate is true. Thus, for example, if `person` is a person, then invoking

```
person.hasA(new BillyBobPred());
```

would return all `Things` with name `billy-bob` that `person` possesses.

The utility of this method is that every class `T` that subclasses `Thing` in the system comes with a static method `isA` returning a `Predicate` that checks whether an arbitrary `Thing` is an instance of class `T`. Thus, again, if `person` is a person, then invoking

```
person.hasA(MobileThing.isA())
```

returns the list of all `MobileThings` that `person` has in her possession. (Note that the list is returned as a `LinkedList<Thing>`, so you will need casting to get the content at type `MobileThing`.)

Write the implementation of the `hasA` method.

What to submit for this exercise: your modified file `Person.java`.

Exercise 3: Implement a Grader (7 pts)

The only missing now is an implementation of graders. Implement a new class `Grader` for this new artifact. Again, give a thought as to what class you want it to extend. It would be nice if graders could move, for instance.

The special thing a grader does is how it responds to an `ask` method. For every completed homework in his possession, it should produce a new graded homework (with the same core name—see the classes `Homework`, `CompletedHomework` and `GradedHomework` for details), put in the same room as himself, and destroy the completed homework. You may want to look at other classes in the game to see how some similar actions are performed. What other artifacts perform similar operations?

Once you have implemented the class, uncomment the code in `Adventure.createWorld` that creates graders.

What to submit for this exercise: your file `Grader.java`

Exercise 4: Roll Your Own (25 pts)

For this exercise, I am asking you to come up with a substantial extension to the game, of your own design. This exercise is worth more than the others; this is the place to demonstrate that you understand the

structure of the code, and that you can apply some of the principles that you learned in the course.

This is your opportunity to be creative. The only hard requirement is that you must add at least one new class to the system.

- You will be graded on the quality and scope of your extension. Thus, if you only slightly adapt an existing class (e.g., implement a homework tracker that looks like a GPS tracker), you will not score high on the scope axis. There are two approaches that you can take: implement one big extension that has a large scope (e.g., implement a full-fledged fighting system, or implement a plot-driven scenario with scheduled events), or many many small extensions that perhaps by themselves do not add a lot on their own (e.g., a teleporter, or a homework tracker), but when put together add up to a lot of new functionality.
- Originality will make me happy, and a happy grader tends to be more lenient.
- You will also be graded on the quality of the documentation that you will provide for your extension.

Get in touch with me (riccardo@ccs.neu.edu) if you want me to assess whether your planned extension is sufficiently scoped.

Feel free to do what you want here—you do not have to stick to the theme of the game as we designed it, so aliens with ray guns and wormholes are perfectly okay.

I will post some ideas on the web site throughout the week in case you get stuck, but you should not feel bound by those either.

What to submit for this exercise: stay tuned; we have not yet figured out the best way to submit your solution.