

Lab: Java Adventure Game

Part 1

This lab was inspired by and adapted from both M. Kolling's Zork lab and MIT's 6.001 Adventure Game. In the first part of the lab, you'll implement simple changes to the basic adventure game. In the second part, you'll allow your programming and creative talents to run wild, expanding the game world in whatever way you choose. So start thinking about what sort of changes you'd like to make.

Keep in mind that we write programs for people to read, and only secondarily for machines to execute. As in all forms of creative expression, your effort is wasted if someone else cannot see the elegance and beauty of your thoughts, design and implementation. You would not attempt to write a paper without an introduction. The same is true for a program. It is essential that you direct the reader of your work by carefully documenting your design in the form of overall comments at the beginning of the program. You would not hide the purpose or theme of your paper and similarly, comments for each method are not only required but essential. Use the style guide to help you.

Writing most of your comments before coding is much like creating an outline for a paper. You can fill in the details later, but the initial comments should point the way. Become adept at this step and you will find that you will spend much less time coding and debugging your programs **and** you will have much more fun. Also, you **will** lose style points if you begin coding before you have commented.

During this first part of **Java Adventure Game**, you may work in groups of one or two. No groups of zero allowed! Groups of more than two require approval of your teacher (which will rarely happen, so you'll need a very good reason!) You are free to choose your own group members from among the students in your class, except that if you are new to the Java language you **must** work with another person who is familiar with the Java language.

1. Putting Harker on the Map

Make a new directory for your work on this lab, and download the following files to it: Command.java, CommandWords.java, Game.java, Parser.java, Person.java, Room.java. Then compile and run the Game class. Try walking around the world. Draw a map showing all the "rooms" and how they're connected.

2. You Can Change the World

Here are some important questions about the game that you should answer in your notebook, before going on.

- A. Open up Room.java. What attributes does a Room have?
- B. What happens when a Room's add method is called?
- C. Now open up Person.java. What attributes does a Person have?
- D. Why does a Person need to know what Room they're in, when a Room already knows what people are in it? What potential danger is there in storing redundant information like this?
- E. What happens when a Person's changeRoom method is called? What would happen if the changeRoom method did not call any of Room's methods?

Check point 1: Call your teacher and discuss your answers!

Open up Game.java, and replace "YourNameHere" with your name.

Create a new room, and connect it to other rooms in the game in a consistent manner. (In other words, if you get there by going north from some room, you should leave it by going south back to the other room.) Be sure to add your room to the rooms list.

Create a new person, and have them start in your new room. Be sure to add your person to the otherPeople list. Test your changes before going on!

Ask yourself the following questions (be sure to answer yourself!):

- 1. What data structures/Abstract Data Types are used in the Game class?
- 2. What particular advantage(s) does the data structures/Abstract Data Types used have?
- 3. What other data structure(s)/Abstract Data Types could you have used?

Check Point 2: Record your answers in your notebook, then call your teacher and discuss your answers.

3. *Who's There?*

Wouldn't it be nice if, when a person entered a room, they greeted to all others in the room? First, we'll need to be able to ask a Room what people are in it, which is a more subtle problem than it sounds.

Suppose an instance of the class `Sender` sends a message to an instance of `Receiver`. It is often helpful to distinguish between the code inside `Receiver` that implements the method, and the code in `Sender` that calls it. We say that `Sender` is a *client* of `Receiver`, because it makes use of the functionality provided by `Receiver`. Whenever we implement a class (such as `Receiver`), we want to make sure that the data used by that class is *hidden* (in private instance variables), so that *client code* (like that found in `Sender`) cannot gain access to the internal workings of our class. The fancy word for this idea is *encapsulation*, and it is one of the most important concepts we'll study in AP Computer Science.

We want to be able to ask a Room what people are in it, so you'll need to go ahead and add a `getPeople` method to `Room`, which should return an `ArrayList` of the people in the Room. However, we don't want `Room` to grant the client access to *the* `ArrayList` it uses internally to store those people. (Otherwise, the client code could change what people are inside the room without calling the appropriate `add` and `remove` methods.) Therefore, your `getPeople` method should make a new `ArrayList`, copy all the people from the Room's `ArrayList` into the new one, and return this new list.

For better or worse, some design decisions have been made for you. And there are implications! Perhaps this design is not the best! While the copy of all the people in the room gets around the problem of allowing a client of the room access to the list of people in the room, it still exposes the people in the room. A misbehaving client could do all sorts of bad things, like changing the state of the people in the room without permission! It's like giving someone access to your Facebook page. This design simplifies what you will do in this step and the next – there are other ways to accomplish this that would be a design improvement. In effect, we are still violating the rules of encapsulation. Think about it now and later you may be called upon to discuss the design and design alternatives.

4. *Hi Everybody!*

The `move` method is called to move a *robot* or *monster* (not the *hero*) into a new room. By changing the `move` method, we can make sure that robots and monsters say hi to any people *that were already in the room when they enter*. In this exercise, you will modify

the move method so that robots greet all people in a room in a single line of output, as shown in the following example:

```
At Shah Hall Bamboozler says -- Hi MrPage MrNikoloff
```

(Combining strings is easy in Java. If `x` points to the `String` "AP", and `y` points to the `String` "PLE", then the expression `y + x` returns the new `String` "PLEAP".)

One of the holy grails of software development is code reuse. If we had just called `println`, we would have to write code to print "At Shah Hall Bamboozler says --". But the `say` method *already does this for us*, and we would prefer to reuse this functionality by simply calling `say`.

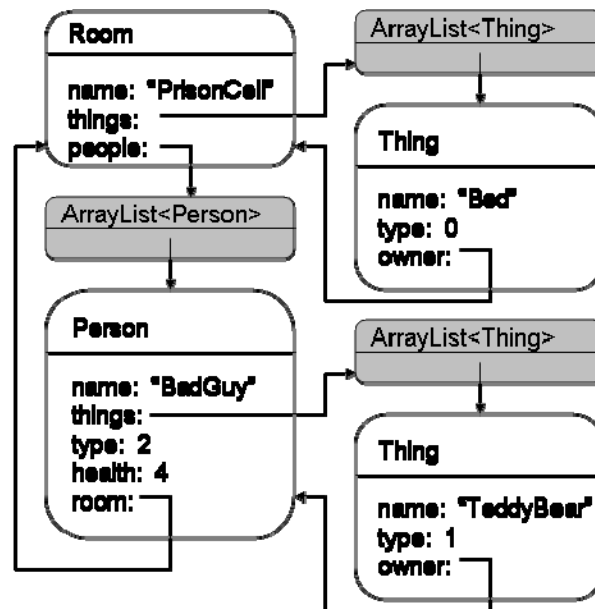
Make sure your people only say hi *when someone else is in the room* and that they *don't say hi to themselves*. Make sure that your people only say hi when they actually move to a different room. Go test your code!

5. The Thing Is...

It'd be nice if there were items in the rooms of our game. To this end, make a new `Thing` class, which might be used as follows.

```
Thing bed = new Thing("Bed", Thing.HEAVY);
bed.changeOwner(someRoom);
Thing bear = new Thing("TeddyBear", Thing.MOBILE);
bear.changeOwner(somePerson);
```

This code creates two instances of the `Thing` class. The following instance diagram shows a heavy `Thing` owned by a `Room` (because the "Bed" is located in the "PrisonCell") and a mobile `Thing` owned by a `Person` (because the "TeddyBear" is owned by the "BadGuy"). Knowing this, what type should you declare `Thing`'s owner instance variable to be?



There will be at least two types of things in our game: heavy things and mobile things. (Later, you may decide to add other kinds of things, such as food, etc.) We will note this by setting the `type` field to 0 for heavy things and 1 for mobile things. But *no one* wants to see 0s and 1s all over our code (except for your computer), or they wouldn't be able to read it! Therefore, we'll define some constants near the beginning of our class definition, as follows.

```
public static final int HEAVY = 0;
public static final int MOBILE = 1;
```

(`public`, because other classes will need access to these variables. `static`, because these variables are attributes of the class—not of any particular instance. And `final`, because these variables are constant, and hence will never vary.)

Now we can (and should!) write `type = HEAVY`, rather than `type = 0`. (Remember one of our objectives is *readable* code.)

Our `Thing` class will need a constructor, which should take in a name and a type. The `owner` field should be initialized to `null`, indicating that no one owns this `Thing` yet.

Our class will also need the following methods.

- `getName`, which should return the name of the `Thing`.
- `isHeavy`, which will return `true` if this is a heavy `Thing`.
- `getOwner`, which will return the object that owns this `Thing`.
- `changeOwner`, which will change the owner to be the given new owner.

Go ahead and implement the `Thing` class.

6. Putting Your Things Away

When you have finished writing your `Thing` class, modify `Room` so as to keep track of an `ArrayList<Thing>` of things in that room. This list should initially be empty. Add a method called `add`, which takes in a `Thing` and adds it to the room. Note that this method has the same name as the one which adds a `Person` to the room. This is ok. Java has no trouble figuring out which method is being called, thanks to explicit types.

Likewise, add a corresponding `remove` method, for removing a `Thing` from the room.

Now let's go back to your `Thing` class and modify `changeOwner` so that the `Thing` is automatically removed from its old room and added to the new one.

We can do this by modifying `changeOwner` to appear as follows. *Make sure you understand this code*, and then add it into your `Thing` class.

```
public void changeOwner(Room newOwner)
{
    if (owner != null)
        //remove the thing from the old room
        owner.remove(this);

    //change owner
    owner = newOwner;

    if (newOwner != null)
        //add the thing to the new room
        newOwner.add(this);
}
```

(Notice again that we're storing redundant information in our objects. A `Thing` knows where it is, and its location knows what `Things` are there. We have chosen to make a `Thing` responsible for keeping this redundant data in sync, but it is not clear that this is the best solution. What else could we have done?)

In the `Game` class's `setup` method, add some `Things` (including at least one heavy and one mobile `Thing`) to the rooms of your game.

Finally, add a method `thingString` to `Room`, which should return a `String` listing all things in the room. (Should this method be `public` or `private`? A good rule of thumb is to declare each method as `private`, unless there is a good reason to grant other classes access to your method.) Be sure to return something helpful if there aren't any things in the room. Then modify `Room`'s `longDescription` method to use `thingString`. When you've made these changes, your game should behave something like this.

```
You are in The Edge.  
Exits: east south west  
No one is here.  
There's nothing here.  
> go south  
You are in Dobbins Hall.  
Exits: north south  
People here: JavaMan  
Things here: InflatableMethod
```

Be sure to test your changes!

7. Finders, Keepers

A person should be able to carry a bunch of things. Modify `Person` to keep track of what `Things` he/she is carrying. Implement the `add` method, which should take in a `Thing` and add it to the bunch of things your `Person` is carrying. You'll also need to implement the `remove` method, which should take in a `Thing` and remove it from the `Person`'s carried things. We'll see shortly how these methods will get called.

Upon encountering a room with a hat, type "take hat", and you'll get the following response.

```
I don't know what you mean...
```

This is because your game doesn't yet know about the "take" command you'll be adding now. At the top of the file [CommandWords.java](#), add "take" to the mysterious array of valid commands.

Again, try taking something in the game, and this time you'll find that the error message has gone away, but nothing has taken its place. This is because you have not yet written code to handle your new command. Modify `Game`'s `processCommand` method to pass the `Command` to a new method in `Game` called `take`, whenever `commandWord.equals("take")`.

Now we'll add the `take` method to `Game`, which should take in a `Command` and have no return value. Temporarily, we'll just stick a `println` here to print out "in take method" (or similar), so that we can test now to see that it gets called when we type "take" in our game.

Having established that the `take` method is indeed being called, let's begin implementing it. First, we need a suitable error message when someone types "take", but doesn't indicate what to take. ("Take what?") Look at other methods in `Game`, and in the `Command` class to see how to test for this condition.

Our next problem is that, if the user typed "take hat" (for example), then all we'll have is the string "hat", and what we really need is the instance of `Thing` with that name. We can solve this by adding a method to `Room` called `getThing`, which will take in the name of a `Thing` and return the `Thing` itself (or `null` if it's not found). Go ahead and write `Room`'s `getThing` method. Note that you need to use the `equals` method to compare two `Strings` (for example, `string1.equals(string2)`).

Some things to consider:

1. When do you use `.equals`?
2. When do you use `==`?
3. What is the difference between the two?

Check Point 3: Call your teacher and discuss the answers (which you have recorded in your notebook) to these three questions.

Now, let's go back to `Game`'s `take` method, and add code to find the `Thing` with the name the user typed. If there's no thing with that name in the room, print a suitable error message. ("It's not here!") Otherwise, let's pass the `Thing` to a new `take` method that we'll add to our `Person` class.

Open the `Person` class, and add a method called `take`, which takes in a `Thing` and does not return a value. If the `Thing` is heavy, the person should merely say "I try but cannot take `BigJavaBook`" (if `BigJavaBook` is a heavy thing in your game). Otherwise, the person should say where they're taking the object from, such as "I take `Fork` from `The Edge`". You will need to modify the `Thing`'s owner to be the person taking it. This requires that you call the `changeOwner` method in the `Thing` class. But there is a problem. The `changeOwner` method expects a `Room` object as a parameter and we want to give it a `Person` object! Whatever shall we do?

The key to our dilemma is polymorphism. Polymorphism is Java's ability to abstract methods and then automatically determine the correct object at run time. It is really an exciting idea that is unique to object-oriented programming. What we will do is introduce the idea of an *interface*. An interface in Java describes all of the characteristics of all classes that implement it. You can think of it as a description of all of the services any class that implements the interface must provide and Java will take care of calling the method in the appropriate class. A Java interface defines only the method headers. You can also define constants, but an interface has no instance fields. Further, all of the methods (and constants) in an interface are `public` by default, and the access level cannot be changed. (In an `Interface`, use of the keyword `public` is optional. If you use the keyword `public` in an `Interface`, it is equivalent to hanging a sign around your neck that says "NOOB".)

Polymorphism comes into play because a Java Interface is recognized as a valid type in Java. If you have the Interface Comparable for example, and the classes Class A, Class B, Class C all of which implement the Interface Comparable, we can do the following:

```
A anObject = new A();
B otherObject = new B();
Comparable polyObject;
polyObject = A;
polyObject.compareTo(Object other); // calls A's compareTo
polyObject = B;
polyObject.compareTo(Object other); // calls B's compareTo
```

The only restriction is that when you point using an interface type, you may only call methods whose headers are defined in the interface.

Add (and comment) the following Java Interface to your project.

```
interface ThingOwner
{
    void add(Thing thing);
    void remove(Thing thing);
    String getName();
}
```

Now, change your Person and Room class so that they implement ThingOwner. For example:

```
public class Room implements ThingOwner
```

Make sure that you do, in fact, implement the methods add, remove, and getName in both the Person and Room class.

Go test your game now, including typing just "take", "take <some object that isn't here>", "take <some heavy object>", and "take <some mobile object>".

When you've completed these changes, make sure that you can now take a mobile object from a room, and that when you return to that room, the object no longer appears there.

Questions to ponder:

1. Who should call the methods add and remove?
2. Who should call the method changeOwner?

Check point 4: Call your teacher and justify your answers (which are recorded in your notebook)!

8. Kleptomania

It seems rather unfair that the hero can take things and robot people can't. In this exercise, we'll modify the robot people so that they'll be able to take things, too. First, we'll need to be able to ask a Room for a random thing.

Find the code that makes robots and monsters exit to a random room. There, you'll see the method `Game.random(int)` being called. Notice that we're calling this method *directly* on the `Game` class, and *not on a particular instance* of `Game`. (This should remind you of our use of `Thing.HEAVY` or `Person.MONSTER`, where we are somehow accessing an attribute of the class itself, rather than calling a method on a particular `Thing` or `Person`.) `Game`'s `random` method, like `HEAVY`, `MONSTER`, and `Game`'s `main` method, has been declared `static`, making it a feature of the class itself. Research this and make sure you understand what is going on!

Use `Game`'s `random` method to implement a method `getRandomThing` in `Room`, which should return a random `Thing` from the `Room`, or `null` if the `Room` is empty.

Then modify `Person`'s `act` method, so that, after moving, robots (and only robots) will call a new method `takeSomething`, which will cause them to pick up a random item from their room. Be sure to make use of `Person`'s `take` method, rather than redoing all your hard work.

9. Stop! Thief!

In this exercise, we'll modify the game so that other people can steal things from you, and you can steal them back. Here's what you'll need to do.

Add a method `getThings` to `Person`, which should return a list of all the things that `Person` is carrying (a new list—not the `Person`'s secret list of things).

Add a helper method `getAllThings` to `Room`, which should return all the things in the `Room`, including the ones sitting around and the ones held by people in the `Room`.

Modify `Room`'s `getThing` and `getRandomThing` methods, so that they use your `getAllThings` method to look through all the things in the `Room`.

You can now start testing your changes by playing your game. Here are some additional changes you'll need to make.

When one person takes something from another person, they should announce whom they're taking from, rather than where they're taking from. Furthermore, the victim should announce what they are losing, and should throw a fit, as this example shows.

*At the gym JavaMan says -- I take A+ from MrPage
At the gym MrPage says -- I lose A+
At the gym MrPage says -- Yaaaah! I am upset!*

Go ahead and test this change.

Unless you anticipated this problem, you'll probably notice that people are taking things from themselves! An easy (but questionable) way to fix this is to remove a person from the room before calling `getRandomThing`, and then adding the person right back. I'm certain you can come up with a better way. In fact, ***you may not remove a person from the room before calling `getRandomThing` and then add the person right back.*** You do know what things you own and you should be able to check if you already have the thing returned by `getRandomThing`. Also make sure you print a suitable error message ("You're already carrying that!") when the Hero tries to take something they're already carrying.

Finally, when a person dies, all their stuff disappears with them. Modify your game so that when a person dies, their stuff remains in the room they died in. Be sure to test your changes. Turn in your completed code to Athena2.

Part 2 Choose Your Own Adventure

For this part of the lab, you are required to work by yourself to create your own unique game. No group projects will be allowed! Make sure that your lab partner has a complete, documented, copy of the code you produced in part 1 and that you have completed all of the checkpoints.

Now you've created a world full of things and thieves, but you can hardly call it a game yet. The time has come to let your creative juices run wild, and to design and implement your own adventure game. First, you should decide what the goal of your game is. It should be something along the lines of: You have to find some items and take them to a certain room (or a certain person?). Then you can get another item. If you take that to another room, you win.

For example: You have to get accepted at Stanford. But first you'll need to get an A in your Comp Sci class, which means picking up the textbook and then tracking down MrPage and answering a programming question. If you answer correctly, he gives you a RecommendationLetter, which you must mail with your CollegeApplication (if you can find it), and then stay alive long enough to pick up your acceptanceLetter. If you're not accepted within a certain time frame, you'll be condemned to spend four years at Harker Community College, a place even darker than the cave.

Or: You are lost in a dungeon. You meet a dwarf. If you find something to eat that you can give to the dwarf, then the dwarf tells you where to find a magic wand. If you use the magic wand in the big cave, the exit opens, you get out and win.

It can be anything, really. Think about the scenery you want to use (a dungeon, a city, a building, etc) and decide what your locations (rooms) are. Make it interesting, but don't make it too complicated.

Put objects in the scenery, maybe people, monsters, etc. Decide what task the player has to master.

Consider adding a way for the player to "win" your game.

Add at least two new commands. (You might need to extend the parser to recognize three-word commands. You could, for example, have a command "give bread dwarf" to give the bread you're carrying to the dwarf.)

Some other things you might try:

- Add a magic transporter room every time you enter it you are transported to a random room in your game.
- Add weapons to your game, so you can fight back against Grendel.
- Add healers to your game, who can increase a person's health who has been injured by Grendel.

For this section of the lab, you must produce a document that describes your game using complete sentences and paragraphs. Your document must contain a UML diagram showing all of the classes and their interactions except for the Game class, CommandWords class, Command class and Parser (See part 5). Then you must create the classes and add JavaDoc style comments before you start to code.

Check point 6: Call your teacher and show off your design talents. You should plan on scheduling a time to do this so that you can avoid the rush and avoid losing class time waiting for check off (which **will** happen if you don't schedule this check point in advance.)

Your grade for this part depends on how well you use what you have already learned as well as how challenging your game is to create. Trivial extensions to the existing game will not earn as many points as a complete game with non-trivial objectives and a clear way to win the game. You should try to reuse as much of your existing code as possible. You must be able to design and implement your game in 2 weeks or less.