**Project 4 - Adventure**

**Adventure Game**

**Resources**: [**CS5044AdventureLibrary**](https://vt.instructure.com/courses/25917/files/1301323/download?wrap=1)**[View in a new window](https://vt.instructure.com/courses/25917/files/1301323/download?wrap=1),** [**CS5044AdventureJavadocs**](https://vt.instructure.com/courses/25917/files/1301324/download?wrap=1)**[View in a new window](https://vt.instructure.com/courses/25917/files/1301324/download?wrap=1),** [**Message.java**](https://vt.instructure.com/courses/25917/files/1301326/download?wrap=1)**[Preview the documentView in a new window](https://vt.instructure.com/courses/25917/files/1301326/download?wrap=1),** and[**MyGameTest.java**](https://vt.instructure.com/courses/25917/files/1301328/download?wrap=1)**[Preview the documentView in a new window](https://vt.instructure.com/courses/25917/files/1301328/download?wrap=1)**

**Goal**

In this program assignment, you will be implementing a text adventure game. You will be working with a number of classes, many of which will be subclasses of existing classes provided for you in the CS5044AdventureLibrary.jar library. You will also be working with interfaces.

Adventure games have been around for a long time, dating back to the *Colossal Cave Adventure*, (see [Colossal Cave Adventure page (Links to an external site.)](http://www.rickadams.org/adventure/), or [DG Jerz's Colossal Cave Adventure page (Links to an external site.)](http://jerz.setonhill.edu/if/canon/Adventure.htm)), but there is more to the history of adventure gaming than this initial spark. In particular, [Infocom (Links to an external site.)](https://en.wikipedia.org/wiki/Infocom) was famous as a computer game producer who made text (and later, graphic) adventure games. Zork, their oldest adventure game, is considered by many to be the most famous of all text adventures, or "interactive fiction games". You can even [download and play (Links to an external site.)](http://www.infocom-if.org/downloads/downloads.html) Zork I, II, and III. There is even an [on-line PHP version (Links to an external site.)](http://thcnet.net/zork/) of Zork you can play in your web browser. You can also read the [definitive history of Zork (Links to an external site.)](http://www.csd.uwo.ca/Infocom/Articles/NZT/zorkhist.html), or read its [entry in Wikipedia (Links to an external site.)](http://en.wikipedia.org/wiki/Zork).

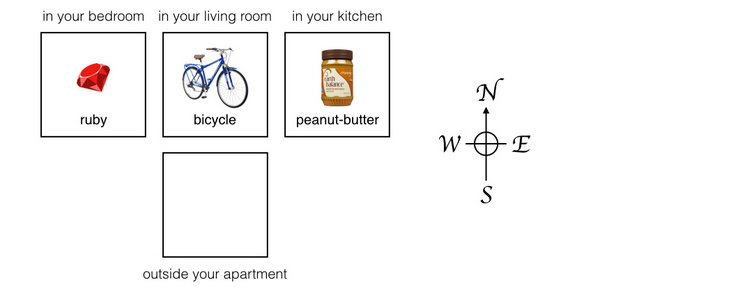
This assignment offers you a large amount of creative freedom, so do not hesitate to have fun with it!

**Learning Objectives**

* Exposure to object-oriented dispatch
* Familiarity with interfaces
* Familiarity with creating subclasses
* Familiarity with overriding methods
* Familiarity with polymorphism
* Familiarity with using library classes

**Lost In Your Apartment**

The game starts off with you in your apartment, and your first goal is to leave. You are in the living room, and going south will take you out of the apartment. But if you try to do that (by entering the command 'go south' or 'south' or even just 's') you get the following message: "You don't want to leave your apartment on an empty stomach." With a little exploration you find that there is peanut butter in the kitchen, which you can eat. Once you have eaten the peanut butter, you can try to leave again, but this time it wants you to lock up your bicycle before you go. Finally, there is a ruby in the bedroom that you have to have with you before the game will let you leave your apartment.



And once you are outside your apartment? Well, that's up to you. You can create more rooms, more objects, even other people to talk to. You're only limited by your imagination!

**Support Classes Provided for You (DO NOT WRITE THESE!)**

Just as in the Tetris assignment, you have a number of support classes available that will help in constructing an adventure game. These classes are all located in the adventure package, which is provided in the CS5044AdventureLibrar.jar file that is linked to at the top of this page (be sure to add it to your project).

The are four main classes for creating adventure games in the package adventure:

* Game represents the main class of a game. It sets the game up, and then enters a loop to read and execute player commands. Note that Game is an **abstract class**: that means you cannot create an object of this class directly, because some of its methods have not been given definitions. Instead, you must create your own subclass (that extends Game) that defines the missing pieces.
* Player represents the person playing the game. The adventure.Player class provides minimal features. However, you can create/extend your own custom subclass of Player that adds any new features you wish.
* Room represents a location in the game. The adventure.Room class provides minimal features. However, you can create/extend your own custom subclass of Room that adds any new features you wish.
* Command is an interface that defines the common properties of all command objects. In particular, it requires every command object to implement an execute() method. There are three classes that implement this interface that are also provided for you in the adventure package: GoCommand, HelpCommand, and QuitCommand. You can use these three commands, and create any new ones you like.

These four classes (well, three classes and one interface) are the aspects of the adventure package that you will use most often. You will probably need to create your own subclasses of each one at some point, although you can work toward a solution incrementally. Study their [**javadoc documentation (Links to an external site.)**](http://courses.cs.vt.edu/%7Ecs5044/api/) well so that you understand what their methods do.

In addition, the adventure package contains two other classes to help you implement a game:

* CommandWords represents a dictionary of known commands in a game. This class uses a [Map (Links to an external site.)](http://docs.oracle.com/javase/7/docs/api/java/util/Map.html) to associate words with Command objects. This makes it easy to add new commands without affecting the structure of this class.
* Parser reads command lines and breaks them up into words, looking the command word up using a CommandWords object. The parser implemented here understands one- and two-word input lines, where a one-word command is a verb (like "quit"), and a two-word command is a verb followed by an object (like "go east" or "take wand").

In addition to these classes, AdventureGUI is an graphical user interface that contains a large rectangular area for displaying text. There is a smaller rectangle below it where you can enter commands, and a "Move" button that you can press to have the command execute (you should also just be able to hit the [return] key).

**Requirements for Your Game Implementation**

You will need to extend several classes to implement a base level of required features in your game. Your game implementation must do the following:

* Include your own custom subclass of Game. It must be called MyGame.
* Include your own custom subclass of Player.
* Include your own custom subclass of Room. The four mandatory rooms are "in your living room", "in your kitchen", "in your bedroom", and "outside your apartment", as shown in the figure above.
* Include a variety of locations/rooms (a minimum of 10 -- the 4 starting rooms + six others).
* It must be possible for a player to reach each of the locations/rooms.
* You must extend the basic game to support **items**. Each item should initially be located in some room. Every room can hold any number of items. Some items can be picked up by the player, others cannot. The player can carry some items with him or her. Every item has a weight. The player can carry items only up to a certain total weight. The three mandatory items are: ruby, bicycle, and peanut-butter. Their starting rooms are shown in the figure above.
* Your game must support the commands take and drop for manipulating items, and inventory (abbreviation: i) for listing out the items currently being carried by the player.
* The player can win. There has to be some situation that is recognized as the end of the game where the player is informed that he/she has won.
* In addition to the "go" command (which is already implemented for you in the GoCommand class), you must also support the following one-word movement commands: north, south, east, west, up, and down, together with one-letter abbreviations for each: n, s, e, w, u, d. Note: you can support all of these with a single command class, so you will loose points if you use six (or twelve!).
* In addition to the commands provided in the adventure package and the additional commands explicitly required in this list, add the following commands:

|  |  |
| --- | --- |
| examine | Allows you to see the description of an object. |
| take | Allows you to take an object from the room and place it in your inventory. |
| drop | Allows you to remove an object from your inventory and place it in the room. |
| inventory | Allows you to view your inventory. |
| eat | Allows you to eat an object that is edible (like the peanut-butter) |
| lock | Allows you to lock up an object (like your bicycle) |

To make it easier to test your game, make use of following Message.java file. Read the descriptions of the methods and make sure that each message is used in the appropriate place in your game. This file is NOT included in the library, so you will have to create a file named "Message.java" in your project and cut-and-paste the code below into your class. Make sure you do not modify the names of the methods in the file. You may be able to make \*small\* changes to the text strings without causing problems with the test cases (but use caution if you attempt this!).

**Message.java** (see resources at the top of this page)

Beyond these requirements, you are free to explore any other game features you wish to provide. Feel free to take advantage of this flexibility to have fun while you complete the assignment.

**Implementing an Adventure Game**

Before you get too far into the design of your adventure game, it will help to play around with the classes a bit and try out a few exploratory changes. This will help you get a feel for making small extensions to the existing features in a way where you can try out each change as you go. Then, as you pursue your design in more detail, you always can keep in mind the concept of **incremental development**--adding one piece of your design at a time, and making sure it works as intended before moving on to the next.

To see how to start creating your own game, let's work through a trivial example. First, create a new Game subclass:

import student.adventure.\*;

public class BoringGame

extends Game

{

public BoringGame()

{

}

}

We cannot yet create any objects from this class, since the base class Game is abstract--that means there are some methods that it declares, but it does not provide code for them. Instead, we need to provide our own code for them in our subclass. These methods are: welcomeMessage(), createRooms(), and createCommands().

First, let's implement welcomeMessage() in our game subclass. This method just determines the message printed when the game starts up:

public String welcomeMessage()

{

return

"<p>Welcome to The World of Simplicity!</p>"

+ "<p>Type 'help' if you need help.</p>"  
 + "<p>Hit [return] to continue...</p>"

; }

Next, let's decide on our map. We can use a couple of rooms from the Virginia Tech campus as an example:

public void createRooms()

{

*// create the rooms*

Room outside = new Room("outside McBryde Hall");

Room lab = new Room("in a computing lab");

*// initialise room exits*

outside.setExit("south", lab);

lab.setExit("north", outside);

*// the player starts the game outside*

player().setCurrentRoom(outside);

}

Finally, we need to add some commands. There are already three commands provided in the student.adventure package that we can use.

public void createCommands()

{

CommandWords commands = parser().commandWords();

commands.addCommand("go", new GoCommand());

commands.addCommand("help", new HelpCommand(commands));

commands.addCommand("quit", new QuitCommand());

}

Now try compiling your game subclass. You can run your game by adding a main() method to your game subclass. The main() method should simply create an instance of your game subclass and then call its play() method. Type your commands into the terminal window. It's pretty boring, but it will get you started.

You should now be able to play your game from a windows command prompt window, or by running your game subclass in Eclipse.

Now that we've discussed the basics of implementing a rudimentary game, you should probably "warm up" by making some basic changes and extensions. For example, you can:

* Change the description of a location to something different.
* Change the exits of a location: pick a room that currently is to the north of another room and put it to the west, for example.
* Add a room (or two, or three, ...).
* Add a new command: say, the command "north" that makes you walk to the north.

Exercises like this should help build your confidence in how the library classes operate, and familiarize you with the basic mechanics of implementing a game. Next, map out your game design--draw yourself a map of your world, write up a list of objects you want to include in the game, and jot down a list of puzzles or tasks you want the player to solve. You can also come up with a list of commands you wish to add--both those that are part of the base-level requirements, and additional commands you want to add in order to support your puzzles or one of your objects.

In implementing your game, the world map is a great place to start. Work on implementing the full world map you've worked out, which should be quick and easy. Afterward, begin prioritizing your list of commands. You can implement them one at a time, and try each one out. Remember to write test cases for each one as you go. You might consider implementing all the commands that do not involve objects first, and save those that do involve objects until you've implemented some way for players to carry things, (and rooms to hold them).

***Proceeding incrementally and testing as you go is a big help with an assignment this large***. It helps to manage the complexity of the task, and also gives you some degree of assurance that what you've completed so far really works--and that when you added a new feature, you didn't break any of your past work.

When you do move on to implementing game objects, you will probably need to create your own subclass of the Player class to your project that extends Player. Using your new class in your game is easy too. Just modify your constructor a little to provide the player and parser objects you want to use up to the Game class constructor:

public BoringGame()

{

*// Use alternate superclass constructor that takes*

*// the player and parser as parameters:*

super(new MyPlayer(), new Parser());

}

This code snippet assumes a player subclass called MyPlayer, but your class names are hopefully more meaningful.

**Testing Your Game**

As with other work in this course, you are responsible for writing tests for all the code you write. In this case, because we have separated the GUI interface from the command processing, we are going to ignore the GUI completely and just test that the commands give you the results in your game that you need. The following JUnit test starter file sets up your tests so that you can pass multiple commands to a private method (executeMoves), and then check if the resulting room description and message are what you expect them to be. Use the long room description when you want to check whether a room has something in it.

**MyGameTest.java** (see resources at the top of this page)

Finally, note that you are expected to (or required to) test the main() method of your game subclass. However, if you've incrementally tested all your other methods individually, you really only need one very simple test for the main() method--its so short, it shouldn't need anything more than that. You also must write appropriate test cases for all other public methods in all classes that **you** write, of course.