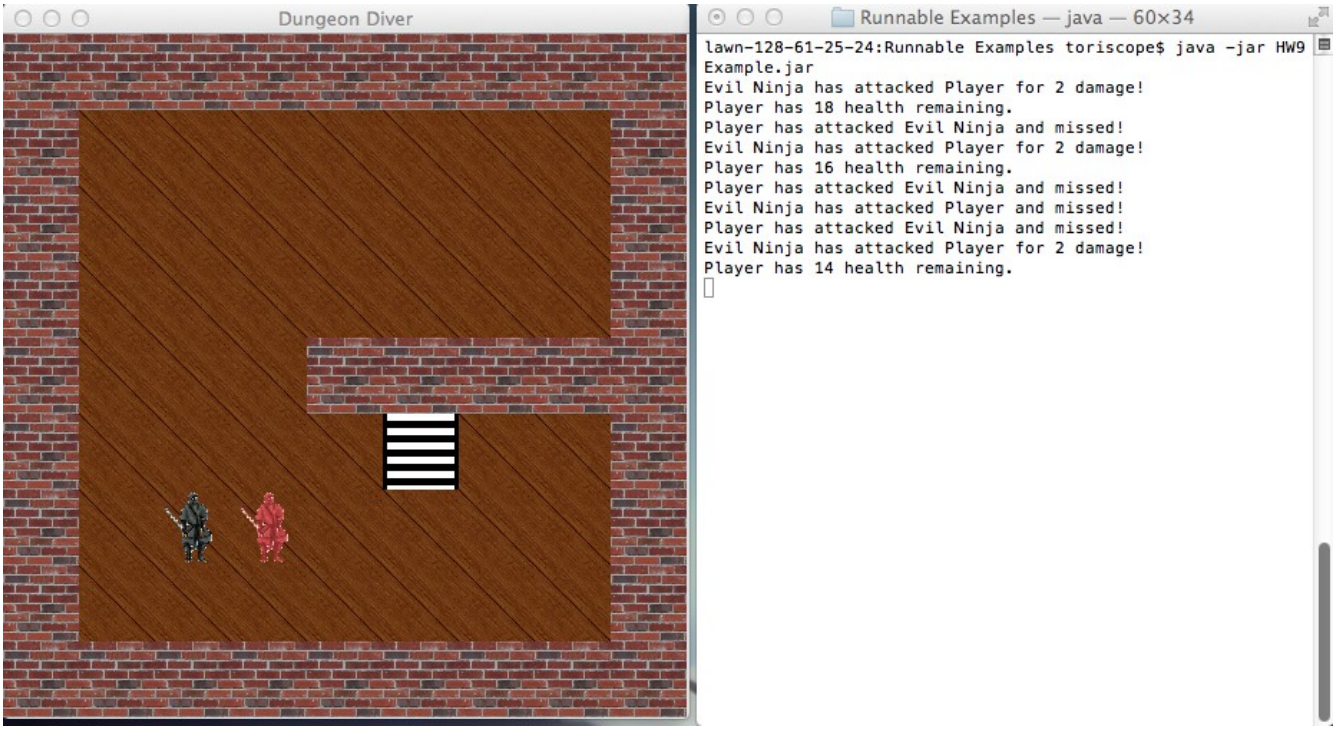
CS1331 Homework 9

Dungeon Diver

Due Friday November 9th, 8PM.



In this homework you will be implementing features of a ninja adventure. The rules of the game are simple, and will make use of hierarchies and polymorphism. To help you follow along with our requirements, we have included a working example of the completed assignment, along with this guide. *Please note: the included example does not contain the additional tile and 2 additional enemies. It also contains multiple levels, a feature that you don't need to implement*.

Please read the whole guide thoroughly before you begin. If you have any questions, please ask them first on Piazza. Check Piazza often, as we offer clarifications and advice there.

The Big Picture

You will be be making a game where a player moves a character around a maze, attempting to reach an exit. The player may encounter enemies, which will attack the player. The player needs to be careful, since they start the level with a certain amount of health, and once that runs out, they game is over. The player can attack the enemies as well, and they also have a set amount of health. For more clarification, run the included example from *command line* (there is valuable output in terminal).

Functionality Requirements

- A randomly generated map of tiles, including **Walls**, **Floors**, an **Exit**, and **one additional tile** of

your choosing. It must effect the player in some meaningful way (like a trap tile, that injures the player if they step on it). Generating a random map is very tough, so you have been given *documented* java code to do it for you. You may use your own generator code, or modify the code freely.

- A keyboard-controlled character that represents the player. The player starts off with a set health and a set attack ability. When the user presses an arrow key, the player character attempts to move in the proper direction to a new tile: if the tile is a Floor tile, and has no enemy on it, then the move will occur. If there is an enemy on the tile, then attempting to move there will initiate an attack on the enemy. Attacking is simple: the enemy has their health lowered according to the "attack" value of the player. We'll talk more about attacking in just a bit. Pressing a key marks the end of the player's turn, so all the enemies will get to move once the player has moved. Watch out!
- Enemies should populate the level. An enemy has a set health and attack, and attempts to move toward the player during each turn. If the player attempts to move *onto* the player, it initiates an attack. The attack rules work just like we discussed in the section above. If the player finds itself completely out of health, it will die and the game is over. Similarly, the enemy is removed from the map when its health runs out. You should make two different enemy types. The changes should be significant, such as a different name, higher damage and/or faster movement.
- Each attack, by the player or an enemy, should have a random change of failing. 50% chance of failure is a good value to choose, but you may pick your own value (as long as it is not 100% or 0%).
- An Exit tile should be somewhere in the level. If the player is on the Exit tile, they win the level, and the game ends in victory.
- The tiles of the level, as well as the characters on the map, will be drawn to a GUI. When something happens (like a character attacks another), you will output the information to the terminal.

Design Requirements

So how should we represent these requirements in code? You are free to implement them any way you like, but you must use *at least* the **hierarchy** we have specified below. If you diverge from the guide, make sure to document it in your javadoc, along with the reasons why you did so. Read the whole design guide before focusing on the details.

- **DisplayPanel.java**: Tasked with rendering the Tiles, and the Creatures inhabiting them.
  - Has a KeyListener, passes along the commands to the Dungeon, including moving the player.
- **Dungeon.java**: This will represent a dungeon, or a random 2D layout of Tiles, filled with creatures. The dungeon object acts like a container for the tiles, and handles moving objects from one Tile to another. It can also be handy to store a reference to the player character. Once the player is done moving or attacking, then every single Tile and the Creature inside it (if it has one) needs to have their update method called.
  - Should have a getter for the Player object.
  - Should have a getter for a Tile, given a tile position in the 2D array.
    - *public Tile getTile(Point tileIndex)*
  - The Dungeon should make a call to the random map generator, and use the resulting

floor-plan to build the 2D array of Tiles. Then, it should replace a random floor tile with the exit tile, and place the enemies and the player in random, appropriate tiles.

- **Tile.java**: This abstract class represents a Tile. Tile has some general functionality that every Tile has in common:
  - A reference to the Dungeon object. This will be useful for issuing commands to the Dungeon (like moving a creature between tiles, in the case of Trap tiles, etc).
  - An Image, representing what should be drawn.
  - A Point object representing it's tile position in Dungeon.java's 2D array of Tiles. It is helpful for a Tile to know its own position in the array, in case you want to add extra functionality later.
  - A Creature (see below) field, where the tile's current inhabitant will be stored. Since, by default, we only have one Creature (player, enemy, etc) standing on a Tile at once, we don't have to make this a list, only a single variable. It should have an appropriate getter/setter.
  - An ***abstract void update()*** method. This method will be called every turn. We will see why we have this method when we start making custom tiles below.
  - A concrete ***draw*** method that takes in a Graphics object, and the tileWidth and tileHeight. Why take in a width and height? When the user resizes the screen, you can make the tiles scale accordingly, like in the provided demo. You don't have to do this, and you may hard-code the width of each tile.
- **FloorTile.java**: Our first specific tile. It will extends Tile, and have its own custom image.
- **WallTile.java**: This tile type will also have its own custom image.
- **ExitTile.java**: This tile will extend **FloorTile**, but add some extra functionality. First, it will have its own custom image. Second, it will override the *update()* method, and do something special: if the creature within this tile is the Player, then we can notify the user that they have reached an exit, and the game is won.
- **CustomTile.java**: You will name this according to its functionality. You must extend Tile or one of its subclasses, and override the update method to do something meaningful.
- **Creature.java:** This abstract class represents a creature, like our player and an enemy. Like Tile, it has a position, an image, and an abstract update method, which the subclasses will fill in via super constructor. It also has a draw method, just like Tile.
  - A private reference to the Dungeon object. This will be useful for issuing commands to the Dungeon (like moving a creature between tiles).
  - A private String field with the *name* of the creature. This can just be the type of creature, or you can get creative and give them a custom name.
  - Two private fields: **attack** and **health**. Attack is the amount of health damage dealt when attacking, and health is the remaining health before this creature will die. These will be set in the constructor.
  - It should have a ***void attack(Creature c)*** method. This method takes in another creature, and initiates an attack, with a small chance of failing (dealing no damage). Regardless, you should print to the terminal How much damage was done, and what the name of the attacker and defender were. You should also print a notification
  - If a creature runs out of health, it should die (be removed from the map). You can put this kind of bookkeeping in the update() method.
- **DungeonElement.java**: By now, you've hopefully noticed that Tile and Creature have much functionality in common. DungeonElement is the new parent class of both of those objects. Look at what variables and methods the classes share, and move the functionality into this

class. These include the image, the tileLocation and the reference to the dungeon, as well as the concrete draw() and abstract update() methods.
    - **Player.java**: The player is a special Creature that the user can control. If this creature dies, the game is over. When deciding if this player can move to a location, you can get the tile from the dungeon object, and use the *instanceof* keyword to check if it is a **FloorTile**.
    - **FirstEnemy.java**: A Creature that seeks to destroy the player. In its update method, it should try and move onto the player (thus attacking).
    - **SecondEnemy.java**: A second enemy of your own design.

Components of Good OOP Design

Here are some basic things we want you to incorporate into your code when coding with hierarchies. Think about how these apply to the hierarchy above.

1. If a class needs something (any resource, or an object of any kind), request it in the constructor of that class. This lets the sub-classes know exactly what they need to provide to the superclass once they extend it. *This doesn't include things that the class would make internally, like a Random object, or a Scanner.*
2. If a variable is to be shared across all instances of a class, such as the **Image** of a *specific* tile, then you should make it static.
3. If multiple subclasses have a functionality or variables in common, you should move them to the superclass.

Drawing Images with a Graphics Object

You can draw an Image using the following method on a Graphics object:

```
graphics.drawImage(image, xPosition, yPosition, width, height, null);
```

The last param, which is null, would be an ImageObserver. You don't need to worry about that. Don't forget to call repaint() on your panel when you want it to redraw everything!

Turn-in Procedure

Turn in the following files on T-Square. When you're ready, double-check that you have *submitted* and not just saved as draft. All .java files should have a descriptive javadoc comment.

Don't forget your collaboration statement. You should include a statement with every homework you submit, even if you worked alone.

Verify the Success of Your HW Turn-In

Practice "safe submission"! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email

almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.

2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
    a. It helps insure that you turn in the correct files.
    b. It helps you realize if you omit a file or files.**
    (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
    c. Helps find last minute causes of files not compiling and/or running.

**Note: Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework (past the grace period of 2 am) will not be accepted regardless of excuse. Treat the due date with respect. The real due date and time is 8 pm Friday. Do not wait until the last minute!

Extra Credit

You may only get a total of +10 extra credit points.

- (+5) The game continues after reaching the staircase. The levels should get larger and contain more enemies.
- (+5) Randomly generated items that the user picks up when they step over them. The item should modify the attack or defense. 2 Different items are required, and at least one must modify the player's health.
- (+5) Make the enemies move on a timer, rather than in response to the player.