# COMP711 Software Assignment
# Text-Based Adventure Game (Version 0)

## Deliverables

1. An Antlr .g4 file containing your GameMap Grammar
2. An Antlr .g4 file containing your PlayerCommand Grammar
3. An archived file containing your IntelliJ Project with source code
4. Log of your software development activities as a PDF file

## Supplied

1. An archive file of an IntelliJ project with starter code

## Brief

Text-based adventure games have been around since the 1960s and have a simple interface in which the player types commands using the keyboard. These commands allow the player to move through and search the gaming *environment*, pick up objects and battle monsters.

For this assignment, you are to individually develop

1. Your own fully functional text-based adventure game engine based on the starting code described below
2. A **GameMap** grammar, using Antlr software to generate a parser for GameMap objects to be input from a text file into your text-based adventure game engine
3. A **PlayerCommand** grammar using Antlr software to generate a parser for player commands
4. A log of containing entries for software development activities you undertake for this project. Each log entry comprises a time stamp and a summary of progress. This can also be done using versioning software such as GitHub using commits

## Gameplay

The player **explores** interconnected rooms picking up food, weapons and valuables to store in their inventory. The player wields weapons in their inventory to **battle** against monsters. The player wins if they find the exit and escape. The player loses the game if their health points reach 0.

The player has health and confidence points which are boosted by eating food or admiring valuables in the inventory. Weapons are used to battle monsters and are stored in the inventory. Chests and other items may be found throughout the maze. Treasure chests contain valuables and are opened with a key and War chests contain weapons and are opened with a lockpick.

As the player moves between rooms, a monster may attack them. The player must **battle** the monster by wielding a weapon and exchanging attacks until one is defeated. During the battle, the player may wield any weapon in their inventory. The player's confidence and wielded weapon determine their attack strength. A monster's attacks decrease the health and confidence points of the player, based on the strength of the monster.

The tables below describe the commands available in **explore** and **battle** modes.

| Explore Mode Commands | Description |
|---|---|
| | When the player enters a room, the following events occur:<br>• A monster may randomly appear (and go to battle mode)<br>• A description of the room is displayed |
| `door n` | Opens door labeled n and enter the room |
| `pickup item` | Pick up an item in room and add to inventory |
| `exit` | Search room to find exit. |
| `describe` | Describes the room, list pickups on the floor and number of doors available |
| `admire valuable` | Admire a valuable pickup in the inventory to increase confidence. The valuable may only be used once to increase confidence, but is not removed from the player's inventory. |
| `eat food` | Eats a food pickup in the inventory to increase health points. Once eaten, the food is removed from the player's inventory. |
| `stats` | Display player health and confidence points and inventory |
| `wield weapon`<br>`wield fistsoffury` | Player wields weapon from inventory for battle<br>Player wields fists of fury (does not appear in inventory) |
| `open chest` | Opens a treasure or war chest in the inventory. The contents of the chest is placed in the player's inventory and the chest removed. |
| `help` | Displays commands in this mode |

| Battle Mode Commands | Description |
|---|---|
| `attack` | Attacks the monster in the room using the wielded weapon |
| `wield weapon`<br>`wield fistsoffury` | Player wields weapon from inventory for battle<br>Player wields fists of fury (does not appear in inventory) |
| `help` | Displays commands in this mode |

## Grammar Designs

You will design and implement two simple grammars for this assignment.

### GameMap Grammar

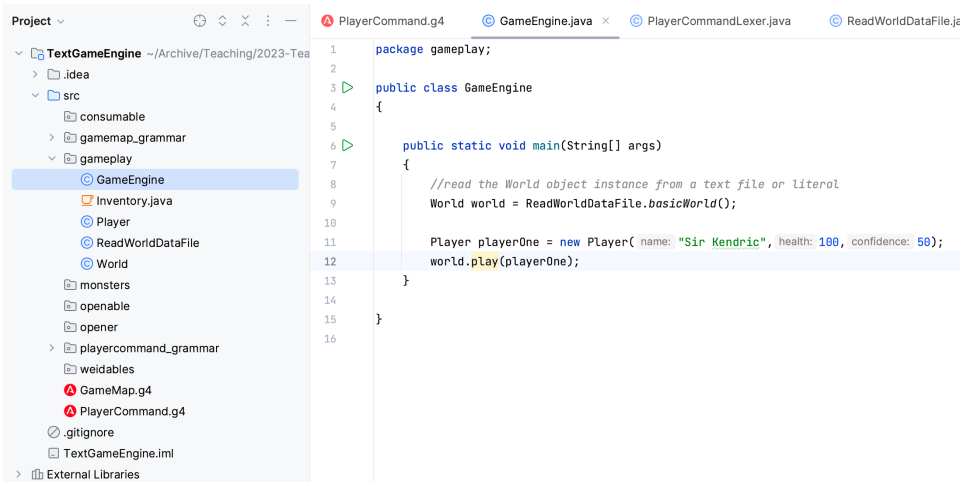
The gaming environment comprises interconnected rooms. The player begins the game in a room designated as the start and finishes the game in an exit room. Design the GameMap grammar with syntactic structures corresponding to the graph structure of the gaming environment, including rules to parse and build Room objects; according to the class description in the next sections.

### PlayerCommand Grammar

The player interacts with the gaming environment according to the explore and battle mode commands. Design the PlayerCommand grammar with syntactic structures corresponding to the structure of the commands in explore and battle mode.

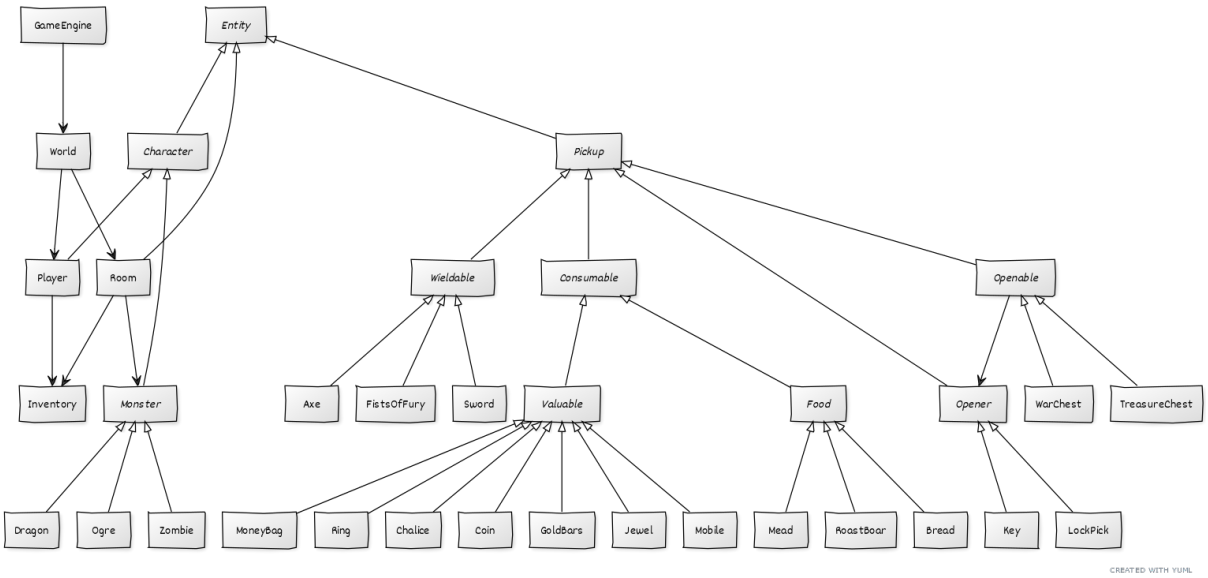
## Getting Started

You have been supplied with an archive file **TextAdventureGame.zip** to import into your IntelliJ workspace. In IntelliJ, go to File -> Open. Click the archive file and then click Finish. The TextGameEngine Project will look similar to:



## Class Hierarchy

The GameEngine class hierarchy comprises many classes. However, the actual coding for each class is very light.




## *Entity* Class Hierarchy

Implement the *Entity* class hierarchy shown in the Class Hierarchy Diagram. Start by defining the abstract class *Entity*. Abstract classes are denoted are italicised in the diagram. All entities in the game will need a description instance variable and a unique identifier instance variable id. Create the Entity(String) constructor which takes a description and sets the unique identifier as

```
this.id = this.getClass().getSimpleName();
```

Add get/set methods for the description. Assume throughout that we will need get and set methods for most instance variables unless stated otherwise. Add a get method for **id**.

The entities in our game will often exhibit randomized behaviour. Write a protected method which returns a random integer between two numbers x and y (exclusive):

```
return new Random().nextInt(y-x) + x;
```

Next, write the compareID(String) method, which determines the entity's id equality to a given string, ignoring the case. Lastly, the toString method should return the entity's id.

## *Pickup* Class Hierarchy

*Pickup* is an abstract class. It contains a constructor to initialize the description.
The classes *Opener*, *Openable*, *Consumable* and *Wieldable* extend pickup:
Lockpick and Key classes extend *Opener* and have constructors to initialize their descriptions.

The *Openable* class has an instance variable for boolean locked and also has a *Pickup* contents instance variable. The constructor should set these values.
*Consumable* maintains a boolean instance variable to determine if the object has been consumed.

## *Wieldable* Class Hierarchy

*Wieldable* is an abstract class which has two integer value instance variables high and low. The **hit** method returns a random number x such that lo <= x < high. A constructor should initialize all instance variables. There are three subclasses; refer to the class diagram and implement each one, defining the necessary constructor for each class.

## *Consumable* Class Hierarchy

*Consumable* is an abstract class and *Valuables* is an abstract class extending *Consumable* that maintains a number representing the object's value. There are seven subclasses; refer to the class diagram and implement each one, defining an appropriate constructor for each. No new methods or data are needed for these subclasses.

*Food* is an abstract class which stores a number representing health points. Refer to the class diagram for the names of the three subclasses extending Food. They do not require any additional methods or data.

## *Character* Class Hierarchy and *Monster* and *Player* subclasses

The *Character* abstract class has an instance variable to keep track of health points which should always store a non-negative number and a constructor to instantiate all instance variables. *Character* contains abstract methods

- `protected abstract int dealAttackDamage();`
- `public abstract int defendAttack(Character enemy);`

The *Monster* subclass of *Character* and has integer values storing the *probability* of the monster appearing in a room and the amount of *damage* the Monster can deal in an attack. The constructor initializes these values and the description.

The *Monster* subclass overrides the dealAttackDamage and defendAttack methods. The dealAttackDamage method returns the value *damage*+r, where r is a randomly selected value between 1 and 10. The defendAttack method simulates an incoming attack from an enemy character. The method invokes d = enemy.dealAttackDamage() and reduces the character's health by d. The value d is returned by the method to report how much damage was dealt to the character.

The *Monster* subclass has a method boolean appear() to determine whether or not the monster appears in a room. If the monster's health is 0, false is returned. Otherwise, the monster appears with frequency based on its *probability* value. A simple way to implement this method would be to pick a random number x between 0 and 101 (exclusive) and return x<=*probability.*

The *Player* class has instance variables for name, health and confidence points, a Wieldable weapon, and an Inventory. You can add any other instance variables needed.
The *Player* class overrides the following methods:

- int dealAttackDamage(), calculates and returns the damage **d** dealt by the player's attack according to the formula **d** := **h** + **h***c**/100, where **h** is the strength of the user's weapon and **c** is the confidence of the player
- int defendAttack(Character enemy), calculates and returns the damage **d** from the enemy's **d** = enemy.dealAttackDamage and decreases the player's health by **d**. Additionally, the player's confidence is decreased by **d**/2.

## Room class

The Room class contains instance variables to store if the room is the final room (and has an exit so the play can finish the game), and a primitive array **Room**[] of all rooms connected to the room. The Room objects maintains a list of Pickup objects present in the room by way of the **Inventory** class, which maintains an ArrayList. ArrayList is a dynamic array, in which elements can be added and removed. The Room object maintains an instance variable of the monster (or monsters) residing in the room. The Room class will need the default constructor (which initializes instance variables to null values) and a constructor with signature

```
public Room(String description,Inventory pickupsInRoom,
            Room[] connectingRooms)
```

## The World Class

The World class implements the core playable functionality of the text game in the play(Player) method. This method is essentially a while loop to parse and process input from the player depending on the current mode: battle or explore. The while loop continues until the game halts: when the player wins, loses or quits the game.

You will need to write the code for private methods

`processExploreUserInput()` and `processBattleUserInput()`

which prompts the player for input using a **Scanner** and processes it as defined by your PlayerCommand Antlr grammar. Your code should be able to handle errors such as mistyped commands.

## Processing explore-mode commands

Suppose processExploreUserInput reads the text *"pickup JEWEL"* from the player, storing the player's typed input in the variable String input. Use the generated Antlr code to parse this string. You can ask the Lexer object to get the next token from the input; e.g.

Token token = lexer.nextToken();

Should be compared to a list of commands possible in the explore mode, using a switch statement. Since the player has invoke the pickup command, perhaps a private pickup() method in the World class is invoked. This method would determine if a *Pickup* object with an id "jewel" was present in the room's inventory (pickupsInRoom). If so, it would be removed from the room and placed in the player's inventory.

As another example, suppose the player later types admire goldbars in the explore mode prompt and the admire() method is invoked, which carries out the following steps:

1. Determine if goldbars is in the player's inventory (use Inventory's select method)
2. If so, then `Pickup pickup = player.getInventory().select(id)`
3. Use the code `(pickup instanceof Valuable)` which returns true if pickup refers to a Valuable object.
4. Since GoldBars is indeed a subclass of Valuable, we cast `Valuable valuable = (Valuable) pickup;`
5. Then, we are able to invoke `this.player.admire(valuable);`

All other commands are structured similarly.

## The GameEngine Class

The GameEngine class contains a main method which reads a World object from a static ReadWorldDataFile class method. These methods use Antlr to parse Room objects using the GameMap grammar.

```
package gameplay;

public class GameEngine
{

    public static void main(String[] args)
    {
        //read the World object instance from a text file or literal
        World world = ReadWorldDataFile.basicWorld();

        Player playerOne = new Player( name: "Sir Kendric", health: 100, confidence: 50);
        world.play(playerOne);
    }

}
```

## Possible extensions to the game

According to the marking rubric, to receive a grade in the A range, you will need to extend the gameplay beyond the assignment instructions. Use your imagination! Some possible suggestions:

- Commands to combine items in your inventory
- Extended Battle commands to de-escalate battles
- New and interesting items
- Teleportation, randomisation of the Player's position in the maze
- Your idea here

## Marking Rubric

| Criteria: | Weight: | Grade A Range 100 ≥ x ≥ 80% | Grade B Range 80 > x ≥ 65% | Grade C Range 65 > x ≥ 50% | Grade D Range 50 > x ≥ 0% |
|---|---|---|---|---|---|
| **GameMap Grammar Implementation** | **50%** | Concise syntactic structures are used to define the gaming environment. Map graph structure and room contents/ properties can be specified by an input file. Interesting extensions to the grammar are created. | Concise syntactic structures are used to define the gaming environment. Map graph structure and room contents/ properties can be specified by an input file. | Problems in reading game environments but mostly works. | Absent functionality for parsing World data or code does not compile |
| **Entity Class Hierarchy Implementation** | **20%** | All classes present with appropriate structure and method overriding | Some minor issues with hierarchy, constructors | Code not appropriately overridden in classes | Absent classes or functionality or code does not compile |
| **PlayerCommand Grammar and Game Play** | **30%** | Excellent handling of text and command entries using the PlayerCommand grammar. Data structures are consistently maintained during gameplay. Interesting extensions of the grammar are created. | Excellent handling of text and command entries using the PlayerCommand grammar. Data structures are consistently maintained during gameplay. | Problems handling commands in either gameplay modes | Absent functionality for parsing text or commands or code does not compile |

## Javadoc Commenting

Please comment your code with your name

```
/**
  * Comment describing the class.
  * @author yourname studentnumber
 **/
```

## Submission Instructions

Submit an archive of all required deliverables to Canvas before the due date.