Technical Report

A detailed document showcasing how each of the game's elements work and how they're achieved using only base Java.

Each of the game's core aspects, classes,methods, parameters and how they're handled will be explained below in detail.

This document was redacted by all team members. Each part was redacted by its respective programmer.

Name of the Project: Project M.Y.N.Y *(My Yard Not Yours)*

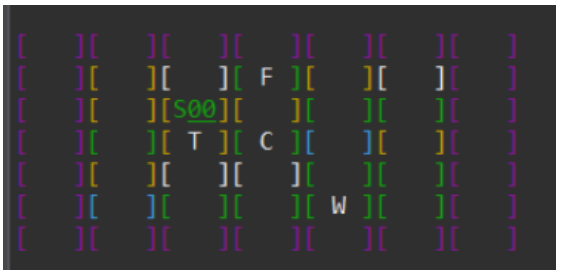
Name of Participant	Slimani Yahia	Degheb Mélissa	Brihmat Neyl Mohamed Tarik	Yudas Djenadi
Group	2	2	2	2
Section	2	2	2	2

Submission Date	31 December 2025
Year	2025/2026
Nom du Prof.	Mr. Abdellahoum Hamza
Faculty	USTHB

Game Concept

Project M.Y.N.Y is a strategic console-based video game built around resource management, territory acquisition, and unit deployment and control. The player's goal is to survive as long as possible with the tools available to them.

The game takes place on a tile-based map with the player's Command Center situated in the middle. Players have a limited amount of turns to construct buildings with gold in order to generate resources. These resources (alongside the presence of a training ground) will allow one to generate units that they can move across the map.



After a certain amount of turns has passed, enemy units will spawn around the edges of the map, and will try to make their way to the Command Center. The player must use their units to vanquish all enemies before they get to their Center. The game ends once the Command Center has been destroyed.

More information on how the game is played can be found in the user manual.

Tools Used

The game was programmed with the Object-Oriented Programming Language Java, as a game of this caliber can be made a lot more effectively by splitting each component into its own object. Certain team members preferred to use the Eclipse IDE for programming, while other resorted to using VS Code

Examples of classes include:

- Abstract classes such as "Element" that contain global parameters including X and Y positions that both Buildings and Units can inherit from.
- Normal classes for concepts like different unit and building variants as well as map terrain
- Static classes for Main, the Map and Resources, which do not necessitate individual objects.

The game uses interfaces, such as a ConsoleColors interface which gives any class that implements it the ability to print text with different colours.

The game uses collections for multiple things, the main ones are:

- Storing resources in a hashmap and referring to them easily. Hashmaps are also used for the random generation of the map at the start of every game.

- Storing buildings and units in lists, as they would otherwise be hard to manipulate with just the map array.
- Lists are also used to determine who a Unit should attack if there are multiple opposing units around it. This is done by storing potential targets in a list and prompting the player to choose one of them.

Table of Contents

- Title Screen
- Elements
- Console Colors
- Map and Terrain
- Buildings
- Units
- Resources
- Main and Game

Title Screen:

Main programmer: *S.Yahia*

Before we start talking about more intricate concepts (including ones in the very class we're starting with), we'll first go over how the Title Screen of the game works.

```
public class Main extends Game implements ConsoleColors {

    public static int turnsBeforeFight = 10; //turns before enemies spawn
    public static boolean gameOver = false;

    static public void main(String[] args) {
        titleScreen(); //procedure that contains what happens on the title screen, comment it out for quick start of game
        game();//will start the game
    }
```

The title screen is a simple method that is set to execute inside the Main class, as it's the one that handles all things related to program execution.

All it does is print the logo and ask for the user's input with a scanner and a switch case.

As simple as this is, there are two things that are worth mentioning about it:

- The game uses a custom method scanInt to get number input. This is to make sure that the user hasn't accidentally entered a string, and prevents the game crashing due to an inputMismatch error.

```
public static int scanInt() {//scan ints and ma

    int y = 0;
    boolean isNum;
    do {
        String x = sc.nextLine();
        try {
            y = Integer.parseInt(x);
            isNum = true;
        } catch (NumberFormatException e) {
            isNum = false;
        }
    } while (isNum == false);

    return y;
}
```

- The logo of the game (looks strange inside the file, but appears normally ingame. That is because it uses backspaces, which cannot be drawn normally as they are special characters, and must therefore be preceded with backspaces themselves.

If the user enters either 1 or 2, game() method will be prompted, starting the game. Choosing 3 will simply prompt the System.exit(0) method, ending the program.

Before we delve further into main and the Game class, we will first go over the different classes that can be found in the game, as both main and game make heavy use of them

The game's main aspects can be split into these categories:

Element:

Main programmers: *D. Mélissa, B. Neyl, S. Yahia*

```
package map;

public abstract class Element {
    public int x, y, hp, basehp, id;

    public String type;//for building and unit names
    public String elementType; //is either "Building" or "Unit"
    //Added because it would otherwise be really hard to tell w
    public String icon = " _ ";

}
```

"Element" is an abstract class that is part of the "Map" package. It's a parent class to both buildings and units and contains parameters that are shared between the two. Buildings and Units both being elements will be an important factor when manipulating the two classes going forward.

This class's parameters along with all other classes' parameters and methods going forward have been made public, as there is no need to worry about security and privacy in a local single player experience.

This class's parameters are:

- x and y, which are integers that represent the element's position on the map.
- hp, which stands for the element's health. Reaching 0 hp means being removed from the game, but this was chosen to be implemented in the Main function rather than a method in this class due to how Buildings and Units will

be stored during the game. More on that later. basehp is the amount of hp an element starts with, and is also the amount healed at the end of a level.
- type, which is how the element will be called ingame (For example, the Command Center's type would be "Command Center").
- Element type, which is used to distinguish between Buildings and Units when going through Element databases.
- icon, which is the element's icon on the map. More on icons in the map and terrain section.

This class does not have any methods.

Console Colors:

Main programmer: *This class was found in this Stack Overflow post where a user asked how they could incorporate Colours into their application's console.* *Link to the original post.*

```
public interface ConsoleColors {
    // Reset
    public static final String RESET = "\033[0m";  // Text Reset

    // Regular Colors
    public static final String BLACK = "\033[0;30m";   // BLACK
    public static final String RED = "\033[0;31m";     // RED
    public static final String GREEN = "\033[0;32m";   // GREEN
    public static final String YELLOW = "\033[0;33m";  // YELLOW
    public static final String BLUE = "\033[0;34m";    // BLUE
    public static final String PURPLE = "\033[0;35m";  // PURPLE
    public static final String CYAN = "\033[0;36m";    // CYAN
    public static final String WHITE = "\033[0;37m";   // WHITE

    // Bold
    public static final String BLACK_BOLD = "\033[1;30m";  // BLACK
    public static final String RED_BOLD = "\033[1;31m";    // RED
    public static final String GREEN_BOLD = "\033[1;32m";  // GREEN
    public static final String YELLOW_BOLD = "\033[1;33m"; // YELLOW
```

In order to make the game easier to look at, ANSI Escape Codes (which are compatible with most recent versions of Eclipse's console) have been incorporated to add color to the other pure white / back text, depending on your software's theme.

If a class implements ConsoleColors, typing [DESIRED COLOUR] such as RED or GREEN will paste a string which will turn the text into that desired colour when printed. It's important to note that RESET must be used afterwards in order to turn the text back to its default color.

Map and Terrain:

Main programmer: *S. Yahia*

As the core concept of the game involves manipulating elements on a map, it was important to build a system that allows for easy manipulation of such elements. To achieve this, the game uses 2 classes:

- The Map class, which is a static class that holds an array where each cell represents a terrain with multiple properties.
- A Terrain class, which is a cell of that same Map array.

Java being an OOP language has made it easy to build such a system, as we can easily store objects in an array that way.

We'll start with the Terrain class. Some of its properties include;

- type, which determines if the terrain is Grass, Water, Forest or Mountain.
- icon, which is the current icon of the tile on the map.
- color, which is the colour of the tile depending on its type.
- bonus, which will be applied to a building that meshes well with the cell's terrain type.
- isAccesible, which is used to denote some terrain type as inaccessible to player units, such as Water.
- And finally, currentElement, which is a list that will hold a single element at a time. This is how elements will be stored inside the map's cells.

Terrain has a lot of important methods that will prove crucial when manipulating units later, some of which include:

```java
public void setTypeRandom(){
    LinkedHashMap<String, Double> possibleTypes = new LinkedHashMap<>();
    Double proba = Math.random(); // from 0.0 to 1.0
    Double addedTo = 0.0;
    // Weighted probabilities, it's a map containing terrain types with their probabibility of appearing
    possibleTypes.put("Grass", 0.4);
    possibleTypes.put("Mountain", 0.25);
    possibleTypes.put("Forest", 0.25);
    possibleTypes.put("Water", 0.1);
    //All probabilities must add up to 1
    System.out.println(proba);
    for (String i : possibleTypes.keySet()) {

        if (proba <= addedTo + possibleTypes.get(i)) { // 1. proba <= 0.0 + 0.35 // 2. proba <= 0.35 + 0.3 etc.
            setType(i);
            break;
            }
         else {
            addedTo += possibleTypes.get(i);
        }

    }
```

- The setTypeRandom method, which will randomly set the terrain to a random type using weighted probabilities. This was achieved by manipulating hashmaps, where each key represents a type and each value is the probability of it appearing. This is done so that the number of inaccessible water tiles will be small compared to the other common tile types. It also makes sure that the Water type is inaccessible to player units.

```
public void placeElement(Element elem, int placerFaction) {
    if (currentElement.isEmpty()) { // can only have 1 element in list
        currentElement.add(elem);

        icon = currentElement.getFirst().icon;
        this.faction = placerFaction;


    } else
        System.out.println("Space is already occupied.");
}

public void removeElement() {
    if (!currentElement.isEmpty()) {
        currentElement.removeFirst();
        icon = "   ";
        faction = Game.NEUTRAL_FACTION;
    }
}
```

- The placeElement and removeElement methods, which put an element inside the currentElement list and assign its icon. Note that you cannot have more than one element in a tile.

```
 public void setColour() { //sets text colour
 switch(type) {
     case("Grass"):
         color = GREEN;
         break;
     case("Mountain"):
         color = RESET;
     break;
     case("Forest"):
         color = YELLOW;
     break;
     case("Water"):
         color = CYAN;
     break;
  }
}
```

- The setColour method, which will be called from the setType method in order to assign a colour to the cell depending on its type. Note how RESET was chosen for Mountain instead of White to make sure that it's visible if the program is run on an interface with light mode.
- There are also common methods for manipulating Elements. Note how the tile's icon depends on the element that currently resides within it.

```
public void placeElement(Element elem, int placerFaction) {
    if (currentElement.isEmpty()) { // can only have 1 element in list
        currentElement.add(elem);

        icon = currentElement.getFirst().icon;
        this.faction = placerFaction;


    } else
        System.out.println("Space is already occupied.");
}

public void removeElement() {
    if (!currentElement.isEmpty()) {
        currentElement.removeFirst();
        icon = "   ";
        faction = Game.NEUTRAL_FACTION;
    }
}

public boolean isEmpty() {
    return currentElement.isEmpty();
}
```

```java
public class Map implements ConsoleColors {
    static final int MAX_CASES = 21; // Must be an odd number
    static public final int CENTER = (MAX_CASES - 1) / 2; // or 26 in algorithmic language
    static Terrain[][] Grid = new Terrain[MAX_CASES][MAX_CASES];// array[n*n] of Terrain
    public static int visibleGrid = 3; // how many cells are visible around the center
    // will grow every level to signify you taking areas



    // ingame, the origin point of the map will be the center, so (0,0) ingame is equivalent to (25,25) in
    // the array (as in, its center)

    static public Terrain getTileFromCenter(int xPos, int yPos) { // so user can type (-1,0) instead of (24, 25)
        return (Grid[yPos + CENTER][xPos + CENTER]);
    }

    static public void generateMap() {// randomly generate all tiles
        for (int i = 0; i< MAX_CASES; i++) {
            for (int j = 0; j< MAX_CASES; j++) {
                Grid[i][j] = new Terrain(); //getTileReal didn't work here oddly
                Grid[i][j].setTypeRandom();
            }
        }
        getTileFromCenter(0,0).setType("Grass");//Forcing type of center to "Grass"
        CommandCenter cc = new CommandCenter();
        getTileFromCenter(0,0).placeElement(cc, 1);
        Game.gameBuildings.add(cc);
    }
```

We will now move on to the Map class. The Map class has a static array named Grid that will hold all the game's tiles. However, only a small amount of those tiles will be visible to the player. The amount of tiles visible to the player will increase as they progress through the game, simulating the acquisition of territory.

Some of its properties include:

- MAX_CASES, which is the amount of rows and columns of the base array (so the amount of tiles would be MAX_CASES²).
- CENTER, which represents the center tile of the array.
- visibleGrid, which is how many tiles the player has access to when counting from the center. So a value of 3 would let them access a 7x7 around the center, while a value of 1 would only let them access a 3x3 area.

The Map class includes methods that will be essential in manipulating elements later on. Some of these include:

- getTileFromCenter(xpos, ypos): Lets you access a tile with the center representing the (0,0) position. Notice how xpos and ypos are switched inside the return function as the first parameter when accessing an array is the rows while the second is the columns, and one typically expects the opposite.
- generateMap: This method will go through every tile in the array, initialise it, and call the setTypeRandom method for it. Making it so that the map is different every time one starts a new game. It will also set the type for the center (0,0) as grass and put the command center there. (more info on the command center the game.gameBuildings list later).

- westNorthLimit and eastSouthLimit simply get the position of the tile the player can see relative to the REAL array. This will be useful for drawing the visible array in the next method.

```
static public int westNorthLimit() { // if Center = 25 and visible grid = 3, returns 22 (leftmost / uppermost visible cell)
    return (CENTER - visibleGrid);
    // this is also how we return fog tiles
}

static public int eastSouthLimit() { // if Center = 25 and visible grid = 3, returns 28 (rightmost / downmost visible cell)
    return (CENTER + visibleGrid);
    // this is also how we return fog tiles
}
```

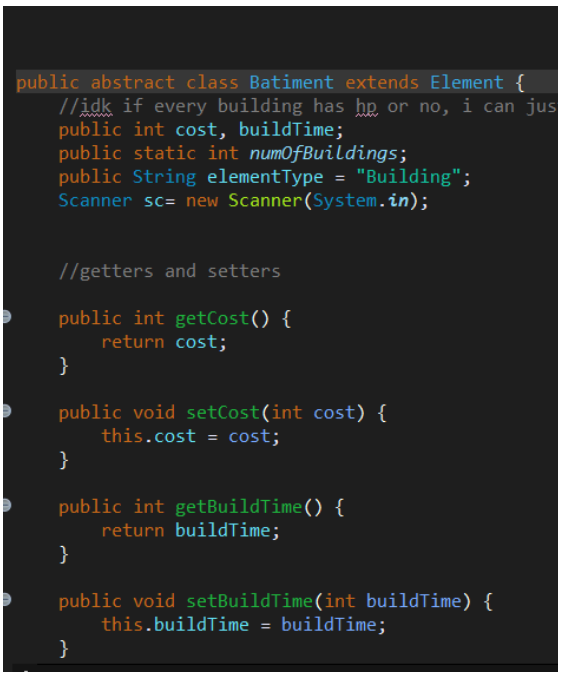
- getMap: The most used method of this class, which prints the visible map to the player in the console. It goes through every visible tile (using the previous methods) and colours it according to certain rules:
    - If the game is still going, it'll colour each tile based on its terrain type, and it'll also print its icon. A tile will be represented by brackets, with the icon in the middle representing the element currently residing within it.
    - If the game has ended (either through loss or victory) all tiles will instead be given a specific color regardless of their current type.

```
static public boolean onFogTile(int x, int y) {//returns true if this position (relative to center) is on the border of the visible map
    return (x == eastSouthLimit() || x == westNorthLimit() || y == eastSouthLimit()|| y == westNorthLimit());
}

static public void getMap() { // print viewable map, will get called each turn
    System.out.print("\n");//can't use println so I do a new line this manually
    for (int i = westNorthLimit() ; i <= eastSouthLimit(); i++) { // "go from the upmost visible row to the downmost"
        for (int j =  westNorthLimit(); j<=eastSouthLimit(); j++) {// "go from the leftmost visible column to the rightmost"
            // Print( "[ " + Icon of current element + " ]") while also coloring the brackets depending on the type of terrain / if terrain is fo
            String color = Grid[i][j].color;
            String iconToPrint = Grid[i][j].icon;

            if(onFogTile(i,j)) {color = ConsoleColors.PURPLE;}
            if (Main.gameOver == true) {//makes map look completely purple with red icons if defeated
                if (Map.getTileFromCenter(0, 0).isEmpty()) {
                    color = ConsoleColors.PURPLE;
                    iconToPrint = ConsoleColors.RED + iconToPrint + ConsoleColors.RESET;
                } else {
                    color = ConsoleColors.GREEN;
                    iconToPrint = ConsoleColors.GREEN + iconToPrint + ConsoleColors.RESET;
                }

            }

            // colours edges of border purple (think of it as fog), this is where enemeies will spawn
            System.out.print(color + "["+ ConsoleColors.RESET + iconToPrint + color +"]"+ ConsoleColors.RESET);
            // will print [ ICON ] for every tile, while coloring brackets according to "color"
        }
        System.out.print("\n"); //new line once we do every column of a row
    }
}
```

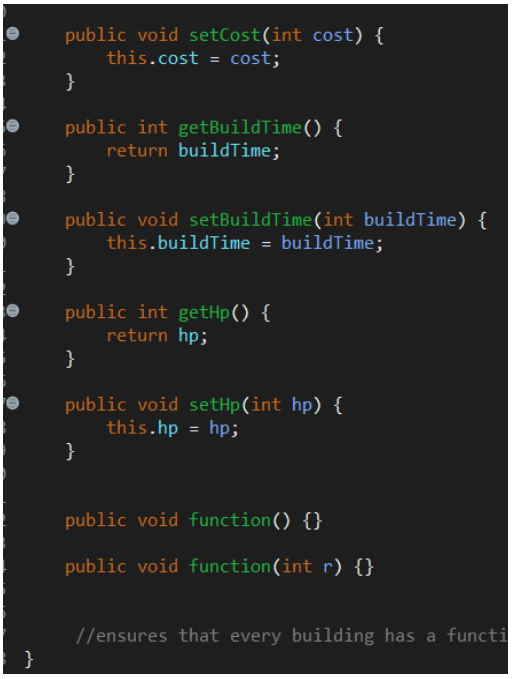
Important part

Buildings:

Main programmer: *D. Mélissa*



```java
public abstract class Batiment extends Element {
    //idk if every building has hp or no, i can jus
    public int cost, buildTime;
    public static int numOfBuildings;
    public String elementType = "Building";
    Scanner sc= new Scanner(System.in);


    //getters and setters

    public int getCost() {
        return cost;
    }

    public void setCost(int cost) {
        this.cost = cost;
    }

    public int getBuildTime() {
        return buildTime;
    }

    public void setBuildTime(int buildTime) {
        this.buildTime = buildTime;
    }
```



```java
    public void setCost(int cost) {
        this.cost = cost;
    }

    public int getBuildTime() {
        return buildTime;
    }

    public void setBuildTime(int buildTime) {
        this.buildTime = buildTime;
    }

    public int getHp() {
        return hp;
    }

    public void setHp(int hp) {
        this.hp = hp;
    }


    public void function() {}

    public void function(int r) {}


     //ensures that every building has a functi
}
```

"Batiment" is an abstract class that is a subclass to the "Element" abstract class, so we import "Element" from the "map" package, it is part of the "batiment" package. "Batiment" is a parent class to both "TrainingGround" and "Generator". The class makes use of the "Scanner" class from the "java.util" package, which is imported to handle user input. As it was previously said, this class's parameters and methods have been made public.

This class's parameters are:

- x and y, which are integers that represent the building's position on the map.
- hp, an integer that depicts the building's health points.
- icon, which is the building's icon on the map, it differs depending on the type of building.
- cost, which stands for the amount of Gold a player has to pay to begin construction, it varies depending on the type of building.
- buildtime, an integer that represents the amount of time a player has to wait for the building to carry its function.
- numOfBuildings, an integer that indicates the number of buildings the player constructs during the game.
- elementType, which portrays how the element will be called during the game, we initialize it to "Building".

This class only has one method:

- function, a void method, it only has one parameter: the integer ‘r’. More on this method below.

Training Ground

```
public class TrainingGround extends Batiment{

    public TrainingGround() {
        this.type = "Training Ground";
        this.elementType = "Building";
        this.cost= 100;
        this.buildTime= 2;
        this.basehp = 50;
        this.hp= basehp;
        this.icon = "T";
        this.icon = " " + icon + " ";
        this.id = numOfBuildings;
        numOfBuildings++;
}
}
```

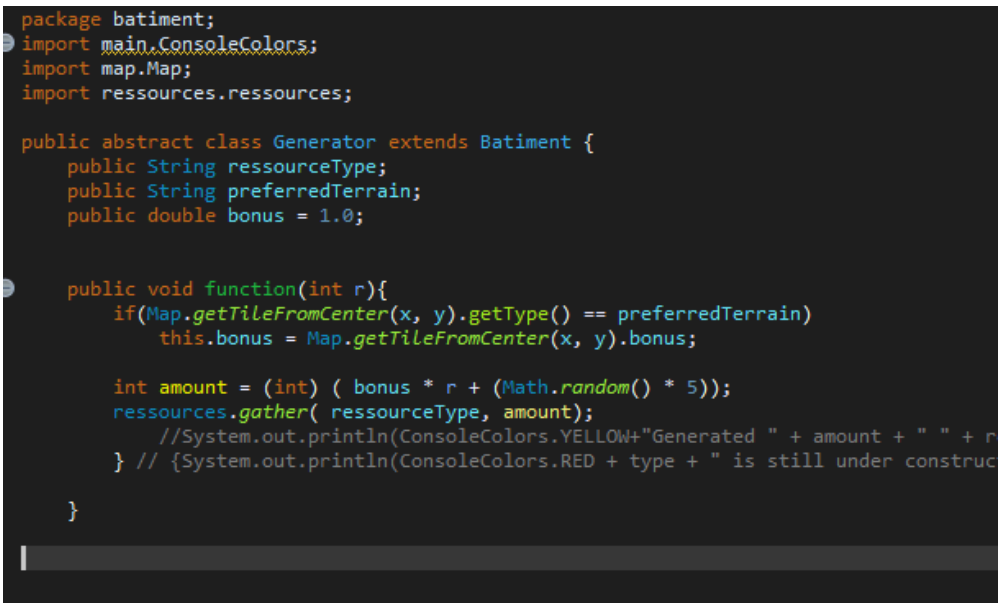
"TrainingGround" is a public class that is a subclass to the "Batiment" abstract class, it is part of the "batiment" package. This class is mainly used as a condition for creating units.

This class contains:

A no-argument constructor: TrainingGround(), it automatically initializes all fields when a new object TrainingGround is created. It contains:

- type, sets it to "Training Ground".
- elementType, set to "Building".
- cost, initializes it to 100.
- buildtime, initialized to 2.
- basehp, it is set to 50.
- hp is initialized to the basehp value.
- icon, initializes it to the value "T". The second line then reformats the value by adding spaces around it.
- id, is set to the numOfBuildings value.
- Increments numOfBuildings.

Generator:



```java
package batiment;
import main.ConsoleColors;
import map.Map;
import ressources.ressources;

public abstract class Generator extends Batiment {
    public String ressourceType;
    public String preferredTerrain;
    public double bonus = 1.0;


    public void function(int r){
        if(Map.getTileFromCenter(x, y).getType() == preferredTerrain)
            this.bonus = Map.getTileFromCenter(x, y).bonus;

        int amount = (int) ( bonus * r + (Math.random() * 5));
        ressources.gather( ressourceType, amount);
            //System.out.println(ConsoleColors.YELLOW+"Generated " + amount + " " + r
        } // {System.out.println(ConsoleColors.RED + type + " is still under construc

    }
```

"Generator" is an abstract class that is a subclass to the "Batiment" abstract class. It is part of the "batiment" package. "Generator" is a parent class to "CommandCenter", "Mine", "Quarry", "Source", "Tree".

This class imports:

- The "main" class from the "ConsoleColors" package, to turn the text into the chosen color when pasting a string.
- The "map" class from the "Map" package, to access its methods.
- The "ressources" class from the "ressources" package, to access the methods it contains.

This class's parameters are:

- ressourceType, a string that represents the generated type of ressource, it varies depending on the type of building.
- preferredTerrain, which stands for the favored type of terrain to construct a certain type of building, it differs depending on the type of building.
- bonus, a double value that represents the bonus a player gets for constructing a building on its preferredTerrain. It is initialized to 1.0, and is set to 1.5 if the building is built on its preferred terrain.

This class's methods are:

function, a public void method which only has one parameter: the integer r. In this method, we have an If condition:

- Using the "getTileFromCenter(x,y)" method and the "getType()" getter from the "Map" package to get the position of the building from the

CommandCenter, and if the type of terrain the building was constructed on is the same as it preferredTerrain, then a bonus of 1.5 more ressources generated is added.

- We also have amount, which is an integer that represents the amount of ressource generated, it is calculated this way:
- We multiply the value of bonus by the amount of ressources generated 'r'.
- We then add a random number between 0.0 and 5.0 to the multiplication using the "Math.random" Java method.
- Using the "gather" method from the "ressources" package, we generate the calculated amount of ressource.

The "Generator" class's subclasses are:

Command Center

```java
package batiment;
import main.ConsoleColors;

public class CommandCenter extends Generator {

    public CommandCenter() {
        this.icon = "C";
        this.type = "CommandCenter";
        this.basehp = 250;
        this.hp = basehp;
        this.x = 0;
        this.y = 0;
        this.cost= 999999;
        this.buildTime= 0;
        this.hp= 250;
        this.ressourceType= "Gold";
        this.elementType = "Building";
        this.icon = " " + icon + " ";
        }
    public void function(int r){
            ressources.gather( ressourceType, Game.gameBuildings.size() * r);
            //System.out.println(ConsoleColors.YELLOW +"Generated " + (int) Game.gameBuildings.size() * r+ " " + ressourceType + "!"+ConsoleColors.RESET );
    }

}
```

"CommandCenter is a public class. It is part of the "batiment" package. The "main" class from the "ConsoleColors" package is imported to turn the text into the chosen color when pasting a string.

This class contains:

A no-argument constructor: CommandCenter(), it automatically initializes all fields when the object CommandCenter is created. It contains:

- icon, it is set to the value "C".
- type, sets it to "CommandCenter".
- basehp, it is set to 250.
- hp is initialized to the basehp value.
- x, which is set to 0.
- y, sets it to 0.
  x and y are both set to 0 so the CommandCenter's position is (0,0), so it is positioned in the middle of the map during the game.

- cost, which is initialized to 999999.
- buildtime, set to 0.
- hp, initialized to 250.
- ressourceType, it is set to "Gold".
- elementType, set to "Building".
- The second line for icon then reformats its value by adding spaces around it.

This class uses the public void method function:

- It multiplies the generated amount of ressources per building 'r' by the number of buildings ingame by using "gameBuildings" and "size" from the "Game" package.
- Then it generates the calculated amount of gold using the "gather" method from the ressources package.

Quarry

```
public class Quarry extends Generator{

    public Quarry() {
        this.type = "Quarry";
        this.elementType = "Building";
        this.cost= 40;
        this.buildTime= 3;
        this.basehp= 40;
        this.hp= basehp;
        this.ressourceType= "Stone";
        this.icon = "S";
        this.preferredTerrain = "Mountain";
        this.id = numOfBuildings;
        numOfBuildings++;
        this.icon = " " + icon + " ";
    }
}
```

"Quarry" is a public class.

This class contains:

A no-argument constructor: Quarry(), it automatically initializes all fields when a new object Quarry is created. It contains:

- type, sets it to "Quarry".
- elementType, set to "Building".
- cost, which is initialized to 40.
- buildtime, set to 3.
- basehp, it is set to 40.
- hp is initialized to the basehp value.

- ressourceType, it is set to "Stone".
- icon, it is set to the value "S".
- preferredTerrain, sets it to "Mountain".
- id is set to the numOfBuildings value.
- it increments numOfBuildings.
- The second line for icon then reformats its value by adding spaces around it.

Source:

```
public class Source extends Generator{

    public Source() {
        this.type = "Farm";
        this.elementType = "Building";
        this.cost= 40;
        this.buildTime= 2;
        this.basehp= 40;
        this.hp= basehp;
        this.ressourceType= "Food";
        this.icon = "F";
        this.preferredTerrain = "Grass";
        this.id = numOfBuildings;
        numOfBuildings++;
        this.icon = " " + icon + " ";
    }
}
```

"Source" is a public class.

This class contains:

A no-argument constructor:

- type, sets it to "Farm".
- elementType, set to "Building".
- cost, which is initialized to 40.
- buildtime, set to 2.
- basehp, it is set to 40.
- hp is initialized to the basehp value.
- ressourceType, it is set to "Food".
- icon, it is set to the value "F".
- preferredTerrain, sets it to "Grass".
- id is set to the numOfBuildings value.
- it increments numOfBuildings.
- The second line for icon then reformats its value by adding spaces around it.

Tree:

```
public class Tree extends Generator{

    public Tree() {
        this.type = "Sawmill";
        this.elementType = "Building";
        this.cost= 40;
        this.buildTime= 5;
        this.basehp= 40;
        this.hp= basehp;
        this.ressourceType= "Wood";
        this.icon = "W";
        this.preferredTerrain = "Forest";
        this.id = numOfBuildings;
        numOfBuildings++;
        this.icon = " " + icon + " ";
    }

}
```

"Tree" is a public class.

This class contains:

A no-argument constructor:

- type, sets it to "Sawmill".
- elementType, set to "Building".
- cost, which is initialized to 40.
- buildtime, set to 5.
- basehp, it is set to 40.
- hp is initialized to the basehp value.
- ressourceType, it is set to "Wood".
- icon, it is set to the value "W".
- preferredTerrain, sets it to "Forest".
- id is set to the numOfBuildings value.
- it increments numOfBuildings.
- The second line for icon then reformats its value by adding spaces around it.

Units:

Main programmer: *B. Neyl*

For the battle system, we have coded units into the game that act as the army or the characters you will use to fight against the enemy.

I started by making the package dubbed “units” which contains multiple classes.

1. Unit.java

These are the variables present in this class:

Variable	Type	What it does
atk	int	Attack of the unit
def	int	Defense of the unit
spd	int	Speed of the unit
range	int	Range of the unit
id	int	ID number of the unit
faction	int	If 1, friendly unit. If 0, enemy unit
basehp	int	HP of the unit at the start of the round.*
aerial	boolean	True: unit can fly over water, false else
numOfUnits	static int	How many units on board, related to ID
cost	Hashmap	Resources needed for training the unit
marine	boolean	If the unit can be placed on a water tile.*

* There exists a hp stat that acts as the current hp of the unit during battle. After each battle, each unit’s hp values go back to their base hp values.

* The marine and aerial units are the same, except that non-aerial units with a range of 1 can not hit aerial units.

For the methods, there are a few basic ones first:

```
public boolean isAlive() {

        return this.hp > 0;

    }
```

This checks if the unit is alive.

```
public void showStats() {

        System.out.println(GREEN + "Health: " + RESET + hp);

        System.out.println(GREEN + "Attack: " + RESET + atk);

        System.out.println(GREEN + "Defense: " + RESET + def);

        System.out.println(GREEN + "Speed: " + RESET + spd);

        System.out.println(GREEN + "Range: " + RESET + range);

    }
```

This shows the hp, atk, def, spd and range stat of each unit.

With all the setters and getters for each variable. No constructor is needed

---

```
public int Attacking(Unit cible) {



        if (cible.isAerial() && !this.isAerial() && this.range == 1) {

            System.out.println("Could not attack aerial unit!");

            return 0;

        }
```

```
        int damage = (int) (this.atk * (0.8 + Math.random() * 0.4)) -
cible.getDef();

        if (damage < 1) {

            damage = 1;

        }

        cible.setHp(cible.getHp() - damage);

        if (cible.faction == 1) {

            Game.updateUnit(cible, Game.playerUnits);

        } else

            Game.updateUnit(cible, Game.enemyUnits);



        return damage;

    }
```

<u>This is the attacking function.</u>

The damage from the attack is calculated so the result will be

(The attacking units attack) * (a number between 0.8 and 1.2) - (The defense of the target)

The number is calculated using a random factor that goes from 0 to 1, which is then added to 0.8 and then multiplied with 0.4. The result would make it so we always get a number that represents 80% to 120% of the attacking unit's attack stat.

```
public void checkForUnits(ArrayList<Unit> potentialTargets){//Checks
around unit if there are opposing units to attack

        int leftmost = x - range;
```

```
        int rightmost = x + range;

        int upmost = y - range;

        int downmost = y + range;

        //this is the attack square around the unit. It should attack
any enemy who is in that square

        //if range = 1, it will check in a 3 by 3 square around the
unit

        //if range = 2, it will check in a 5 by 5 square around the
unit



        for (int i = upmost; i<=downmost; i++) {

            for (int j = leftmost; j<=rightmost; j++) {//Checks around
the unit for enemies

                if (!Map.getTileFromCenter(j, i).isEmpty() &&
Map.getTileFromCenter(j, i).getElement().elementType == "Unit") {//if
tile isn't empty and it contains a unit

                    if(Map.getTileFromCenter(j, i).faction !=
this.faction) {//If the found unit's faction is different than the
attacker


potentialTargets.add((Unit)Map.getTileFromCenter(j, i).getElement());//
add to potential targets

                    }//we use (Unit) to convert from the superclass
Elements to the subclass Unit. It is safe to do so as we made sure the
element found was a unit previously

                }

            }
```

This function is used in other classes of the game (mostly in the game.java class in the main package) to check if a unit is in range of another unit.

```
public int Attacking(Batiment cible) {



        int damage = (int) (this.atk);

        cible.hp -= damage;

        Game.updateBuilding(cible, Game.gameBuildings);

        if (damage>0) {

            System.out.println(this.icon + " dealt " +RED+ damage +
RESET+ " damage to " + cible.type + "!");

            Game.wait(100);

        }

        return damage;

    }
```

This is another attacking function specific to attacking buildings.

```
public boolean moveTo(int x, int y) {

        if(!Game.canAddElem(this.x + x, this.y+y, aerial, this.faction,
false)) {System.out.println("Couldn't move unit.");return false;}

        Map.getTileFromCenter(this.x + x, this.y +
y).placeElement(this, faction);

        Map.getTileFromCenter(this.x, this.y).removeElement();

        this.x += x;

        this.y += y;

        Map.getMap();    return true;
```

This function is used to move a unit from a tile to another depending on its speed stat and if its aerial or not.

```
public void assignIcon(int faction) {

        if (faction == Game.PLAYER_FACTION)

            this.icon = GREEN + icon + GREEN_UNDERLINED +
String.format("%02d", id)+ RESET;

        else

            this.icon = RED + icon + RED_UNDERLINED +
String.format("%02d", id)+ RESET;

    } //String.format("%02d", id) will print the id with in 2 digits,
that way the tile size will be consistent while showcasing up to 100
units at once
```

This function is used to assign the icons of each unit on the visible map, depending on what type of unit they are and what team they are on.

There also are 5 classes dedicated to each unit type, 5 unit types in total.

The most basic one is the soldier unit:

```
package units;

public class Soldat extends Unit {

    public Soldat() {

        this.icon = "S";

        this.basehp = 50;

        this.hp = basehp;

        this.atk = 30;

        this.def = 10;

        this.spd = 1; //speed is how many tiles of the map a unit can
move per turn

        this.range = 1; //range is how many tiles the enemy has to be
away to be attacked

        this.type = "Soldat";
```

```
        this.elementType = "Unit";

        this.aerial = false;

        this.cost.put("Gold", 20);

        this.cost.put("Food", 20);

        this.cost.put("Stone", 10);

        id = numOfUnits;

        numOfUnits++;

        if(numOfUnits>99) {numOfUnits=0;}

    }}
```

Each unit has their specific stats and costs. (For details about the stats, see the game manual)

Resources:

Main programmer: *D. Yudas*

```java
package ressources;

import java.util.HashMap;
import main.ConsoleColors;

public class ressources {


    static public HashMap<String, Integer> currentRessources= new HashMap<>();

    static public void addRessources() { // adds all ressources, put it at the start of main
        currentRessources.put("Wood",1000);
        currentRessources.put("Stone",1000);
        currentRessources.put("Gold",1000);
        currentRessources.put("Food",1000);
          }

//      Add a new ressource type
//     public void addResourceType(String type) {
//         currentRessources.putIfAbsent(type, 0);
//     }

    // increase ressource
    static public void gather(String type, int value) {
        if (value <= 0) return;

        currentRessources.put(type, currentRessources.get(type) + value);
    }

    //  remove ressource
    static public boolean consume(String type, int value) {
        if (value <= 0) return false;
```

```java
    // increase ressource
    static public void gather(String type, int value) {
        if (value <= 0) return;

        currentRessources.put(type, currentRessources.get(type) + value);
    }

    //  remove ressource
    static public boolean consume(String type, int value) {
        if (value <= 0) return false;

        int current = currentRessources.get(type);
        if (current < value) return false;

        currentRessources.put(type, current - value);
        return true;
    }

    static public int getQuantity(String type) {
        return currentRessources.get(type);
    }
// prints out the types and values

 static public void showRessources() {
     for (String i: currentRessources.keySet()) {
            System.out.println(ConsoleColors.YELLOW + i  + ": " +ConsoleColors.RESET+ currentRessources.get(i));
        }
 }


//public static void main(String[] args) {
```

The **Resources** class is part of the package named ressources ,responsible for managing all in-game resources. The class centralizes the storage, modification, and display of resources such as Wood, Stone, Gold, and Food. This class uses only **static variables and methods** because resources are global and shared across the entire game,meaning it does not need to be instantiated. All attributes and methods are declared **public**, as the game is a local single-player with no concerns of privacy or security problems.

Variables

- **currentRessources** (`HashMap<String, Integer>`)
  This variable stores all available resource types and their current quantities.
  The key represents the resource name (for example `"Wood"` or `"Gold"`), while the value represents the amount of that resource of which the player currently owns.

A `HashMap` was used because it allows fast access to resources and makes it easy to add new resource types if needed.

Methods

- **addRessources()**
  Initializes the default resources at the start of the game.
  It inserts predefined resource types into the map with an initial quantity.
  it is intended to be called once in the `main` function to set up the starting values.

- **gather(String type, int value)**
  Increases the quantity of a given resource.
  If the value provided is less than or equal to zero, the method does nothing.
  Otherwise, the specified amount is added to the current quantity of the resource.
  This method is used when units gather resources or when buildings generate them.

- **consume(String type, int value)**
  It decreases the quantity of a resource when an action requires it, such as building construction or unit creation.
  If the value is invalid or if the player does not have enough resources, the method returns `false`.
  If the resource is successfully consumed, the method returns `true`.
  This return value allows other parts of the game to verify whether an action can be performed.

- **getQuantity(String type)**
  This method returns the current quantity of a specified resource.
  It is mainly used for checking requirements and displaying resource values.

- **showRessources()**
  This method displays all resource types and their quantities in the console.
  It runs through the resource map and prints each resource along with its

current amount.
Console colors are used to improve readability.

Main and Game:

Main programmer: *S. Yahia with help from all other team members*

Now that we've talked about each of the game's components in detail, we can finally go over to the main gameplay loop.

```
static public void game() { // aka this is where everything happens
    Scanner sc = new Scanner(System.in);
    Map.generateMap();
    print("And thus, your legend begins.");
    ressources.addRessources();
    Map.getMap();
    String commande;//the input from the player
    boolean advancer;

    while (!gameOver) {



    System.out.println("And so ends your tale.\nGAME OVER");
    System.exit(0); //ends
}
```

When the game() method is launched (after selecting 1 or 2 on the title screen), the map is generated along with your resources before being drawn. A "command" string is declared, which is where player input will be stored. The boolean "advancer" is used to determine whether an action taken by the player will let them move on to the next turn or not (for example, looking at your map or current resources will not end your turn, but building something or moving a unit will do so)

The gameplay takes place inside the while (!gameOver) loop. As its close implies, the game will go on until the gameOver variable is set to true, either by winning or losing. After the game's end, System.exit(0) is used to end the program

Let's take a look inside the while loop and see what happens during a game.

```
boolean advancer;

while (!gameOver) {
    System.out.print("\n");
    if(turnsBeforeFight> 1)
        print(RED +turnsBeforeFight+ RESET + " turns left before the enemy arrives...");
    else if (turnsBeforeFight== 1)
        print(RED +turnsBeforeFight+ RESET + " turn left.");

    print("What will you do?");

    commande = sc.nextLine();//get input from player
    advancer = command(commande);//the game class will output accordingly
    //advancer will be set to true if the player's input is a turn ending action, for example placing a building
    //advancer will be set to false if the player's input isn't a turn ending action, for example viewing the map
    if(advancer) {
        if (turnsBeforeFight > 0)
            turnsBeforeFight--;
        turnActions();//actions that are taken at the end of a turn, i.e generating gold
        if(gameOver)
            break;
    }

    if (turnsBeforeFight == 0) {
        print(RED +"THE ENEMY HAS ARRIVED"+ RESET);
        spawnEnemy();
        turnsBeforeFight--;
    }
```

Knowing that turnsBeforeFight is set to 10 before the loop:

- The user is first prompted to enter their desired command. Which will then be used as a parameter for the command method. It will either return true or false depending on the command typed. This will be stored in advancer.
- If advancer is true, then the action taken by the user will end the turn. In that case, turnsBeforeFight gets decreamented, and the turnAction method is called. The loop will be broken if gameOver was set to true during turnActions, ending the game.
- Afterwards, the game checks if turnsBeforeFight is equal to 0. In that case, it will spawn the enemies, who will make their way to the command center.

```
}

if (turnsBeforeFight < 0 && enemyUnits.isEmpty()) {
    System.out.println("The enemy has been vanquished!");
    wait(100);
    levelIncrease();
    turnsBeforeFight = 10;
    Map.getMap();
}
```

- Near the end of the loop, the game will check if the number of enemies has hit 0 during a fight. In that case, a level up occurs, turnsBeforeFight is set back to 10, and the game continues as normal.

Methods such as command, turnActions, spawnEnemy and levelIncrease will become more apparent as we go through the Game class, which links all the previous classes together into a playable game.

```java
package main;
import batiment.*;

public class Game implements ConsoleColors {
    static public int gameLevel = 1, endLevel = 8;
    static public ArrayList<Batiment> gameBuildings = new ArrayList<Batiment>();//current buildings
    static public ArrayList<Unit> playerUnits = new ArrayList<>();//player buildings
    static public ArrayList<Unit> enemyUnits = new ArrayList<>();//enemy buildings
    static public int PLAYER_FACTION = 1;
    static public int ENEMY_FACTION = 0;
    static public int NEUTRAL_FACTION = -1;
    static public Scanner sc = new Scanner(System.in);
    static public int x,y, id;
    static public Unit u;

    static public boolean command(String com) {//true ends turn, false doesn't
        switch(com.toLowerCase()) {
        case("h"):
        case("help"):
            System.out.println("LIST OF COMMON COMMAND");
            System.out.println("map: Shows the current map.\nstats: Shows stats.\nunit: generates unit.");
            System.out.println("build: begins building construction.\nmove: move a unit.\nattack:attack with a unit.")
            System.out.println("Refer to game manual for other commands.");
            return false;
        case("u"):
        case("units"):
        case("unit"):
```

The Game class's parameters are self explanatory, but a few that are worth exploring are the gameBuildings, playerUnits and enemyUnits lists: All of the game's elements will be stored in these to make manipulating them easier. Without them, one would have to go through all of the tiles on the map in a very unorderly fashion, and actions that specifically require going through all buildings and making them carry their functions, for example, would be cumbersome to write.

We will now go through the class's different methods and see exactly how the game works.

command(String com)

Will perform an action depending on what the user has typed using a switch case. Note how the command is turned to lowercase, ensuring that the command is performed even if the user types it in multiple cases. Some important commands include:

```java
case("b"):
case("build"):
    if (!enemyUnits.isEmpty()) {System.out.println("You cannot build during a fight!"); return false;}
    System.out.println("X value of building?");
    x = scanInt();
    System.out.println("Y value of building?");
    y = scanInt();
    System.out.println("What to build? (Training Ground, Farm, Sawmill, Quarry)");
    String building = sc.nextLine();
    return addBuilding(building, x, y);
```

- build, which will let the user build something by inputting its x and y value. Note how the game first checks if there are any enemies before letting the user build. We will go through the addBuilding method in a bit.

```
case("unit"):
    if (!enemyUnits.isEmpty()) {System.out.println("You cannot deploy units during a fight!"); return false;}
    if (!trainingCampExists()) {
        System.out.println("Cannot deploy units without a working training camp.");
        return false;
        }
    System.out.println("X value of unit?");
    x = scanInt();
    System.out.println("Y value of unit?");
    y = scanInt();
    System.out.println("Which unit to deploy? (Soldier, Archer, Eagle, Fish, Giant)" );
    String unit = sc.nextLine();
    return addUnit(unit,x,y,PLAYER_FACTION);
```

- unit works almost the same build, with the only difference that it first checks if there is a working training ground beforehand. Let's quickly go over it:

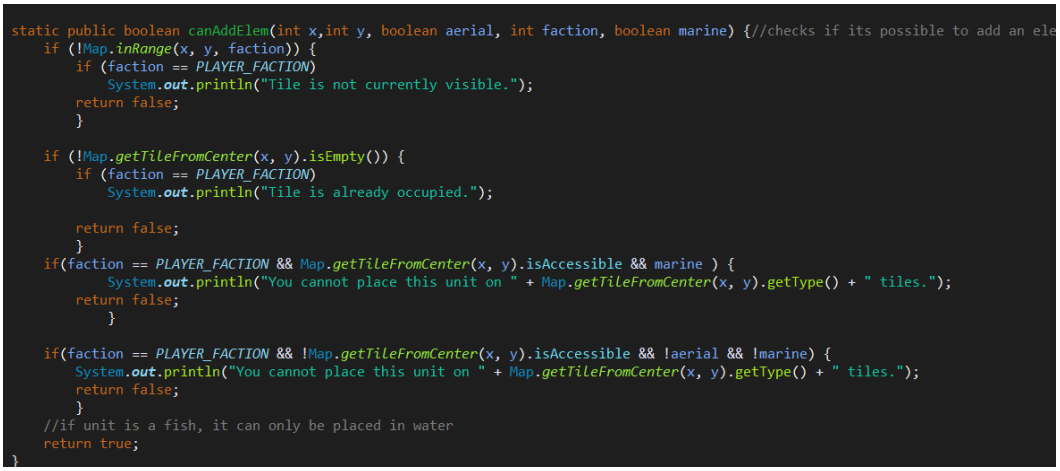
```
static public boolean trainingCampExists() {
    for (Batiment i: gameBuildings) {
        if (i.type == "Training Ground" && i.buildTime <= 0) {
            return true;
        }
    }
    return false;
}
```

This method will go through the gameBuildings list, and check if there is a training ground with a buildTime lower or equal to 0.

```
case("m"):
case("move"):
    System.out.println("Please type out the id of the unit you wish to move.");
    id = scanInt();
    u = findUnitThroughID(id,playerUnits);
    if (u == null) {System.out.println("This id does not exist."); return false;}
    return playerMoveUnit(u);
case("a"):
case("atk"):
case("attack"):
    System.out.println("Please type out the id of your attacking unit.");
    id = scanInt();
    u = findUnitThroughID(id,playerUnits);
    if (u == null) {System.out.println("This id does not exist."); return false;}
    return attackUnits(u);
case("stats unit"):
case("sunit"):
case("su"):
    System.out.println("Please type out the id of the desired unit. (You can also view the stats of enemy units)");
    id = scanInt();
    u = findUnitThroughID(id,playerUnits);
    if (u == null)
        u = findUnitThroughID(id,enemyUnits);
    if (u == null) {System.out.println("This id does not exist."); return false;}
    u.showStats();
    return false;
default:
    System.out.println("Unknown command.");
```

- The move, attack and stats unit commands all work under the same principle: The user is first asked to type out the id of the unit they wish to operate on (Unit ids are 2 digit codes visible on the map at all times. Each unit has a unique id). The game will then check if that unit exists, and carry the action if it finds it.

canAddElem(int x,int y, boolean aerial, int faction, boolean marine)



```
static public boolean canAddElem(int x,int y, boolean aerial, int faction, boolean marine) {//checks if its possible to add an ele
    if (!Map.inRange(x, y, faction)) {
        if (faction == PLAYER_FACTION)
            System.out.println("Tile is not currently visible.");
        return false;
        }

    if (!Map.getTileFromCenter(x, y).isEmpty()) {
        if (faction == PLAYER_FACTION)
            System.out.println("Tile is already occupied.");

        return false;
        }
    if(faction == PLAYER_FACTION && Map.getTileFromCenter(x, y).isAccessible && marine ) {
            System.out.println("You cannot place this unit on " + Map.getTileFromCenter(x, y).getType() + " tiles.");
        return false;
            }

    if(faction == PLAYER_FACTION && !Map.getTileFromCenter(x, y).isAccessible && !aerial && !marine) {
        System.out.println("You cannot place this unit on " + Map.getTileFromCenter(x, y).getType() + " tiles.");
        return false;
        }
    //if unit is a fish, it can only be placed in water
    return true;
}
```

This method will return true if a specific element can be added to a tile of x and y position. It will return false if:

- x and y are outside the allowed / visible range.
- The tile is currently occupied by another element.
- The player tries placing a non-marine, non-aerial element on inaccessible tiles.
- The player tries placing marine elements on accessible tiles.

addUnit(String unitToAdd, int x, int y,int faction)

```
////////////////// UNIT STUFF ////////////////////

  static public boolean addUnit(String unitToAdd, int x, int y,int faction) {// adds unit at pos (x,y) b
  //This is so that the player can add a function just by inputing its name, hence the string
      switch(unitToAdd.toLowerCase()) {
          case("s"):
          case("soldier"):
          case("soldat"):
              return addUnit(new Soldat(), x, y, faction);//function to make code more readable
          case("a"):
          case("archer"):
              return addUnit(new Archer(), x, y, faction);//function to make code more readable
          case("e"):
          case("eagle"):
          case("aigle"):
              return addUnit(new Aigle(), x, y, faction);//function to make code more readable
          case("f"):
          case("fish"):
              return addUnit(new Fish(), x, y, faction);//function to make code more readable
          case("g"):
          case("giant"):
              return addUnit(new Giant(), x, y, faction);//function to make code more readable
          default:
              System.out.println("Not a unit, please try again.");
              return false;
          }
  }
```

This method will add a unit on the map based on a text input, positions and a faction. Note how there is another addUnit method that accepts the actual unit object instead of a String.

addUnit(Unit u, int x, int y, int faction)

```
static boolean addUnit(Unit u, int x, int y,  int faction) {//actually adds the unit to both the map and its respective unit array
    if (!canAddElem(x,y, u.aerial, faction, u.marine)) {//checks if element can be added here instead of the previous function becau
        u = null;//hacky way to delete an object in java, should get eaten by the garbage collector as nothing references it elsewhe
        Unit.numOfUnits--;
        return false;
        }

    if (faction == PLAYER_FACTION) { //only consumes ressources if friendly unit
        for (String i: u.cost.keySet()) {
            if (ressources.currentRessources.get(i) < u.cost.get(i)) {//checks a ressource common to both currentRessources and unit
                System.out.println("You need: " + (u.cost.get(i) -ressources.currentRessources.get(i)) +" of \""+i+"\" in order to d
                return false;
            }
        }
        for (String i: u.cost.keySet()) {
            if (ressources.currentRessources.get(i) < u.cost.get(i)) {
                ressources.consume(i, u.cost.get(i));
            }
        }
    }
    u.assignIcon(faction);//This is mostly so that friendly units can be green and enemy units red
    //must be called before being added to the map or else the icon won't be shown on the map as the old one has already been stored
    u.train(1);//make stats better
    Map.getTileFromCenter(x, y).placeElement(u, faction);
    u.x = x;
    u.y = y;
    u.faction = faction;

    if(faction == PLAYER_FACTION) {
        playerUnits.add(u);
        Map.getMap();
    } else
        enemyUnits.add(u);
    return true;
}
```

This method will actually add a unit to the map. It'll begin by checking if that unit can be placed on that specific map position. If it can be placed, it'll then check if that unit is of the player faction, in that case, it'll make sure the player has enough resources to deploy this unit. Once that's done:

- The method will assign the icon of the unit based on its action using this simple method. (note how String.format("%02D", id) was used to only draw the first two digits of the id.)

```
public void assignIcon(int faction) {
    if (faction == Game.PLAYER_FACTION)
        this.icon = GREEN + icon + GREEN_UNDERLINED + String.format("%02d", id)+ RESET;
    else
        this.icon = RED + icon + RED_UNDERLINED + String.format("%02d", id)+ RESET;
} //String.format("%02d", id) will print the id with in 2 digits, that way the tile size w
```

- It will then improve the units' starting stats based on the current game level using a repurposed train method. This makes it so that newly spawn enemies during later levels can still put up with units that have trained over the course of the game.

```
public void train( int numOfTrainingGrounds) { // improves stats based on cu
    int improveRatio = 10-faction;//makes it so that enemies are a little be
    if(Game.gameLevel <= 10) {
        improveRatio = Game.gameLevel-faction;
    }
    int multiplierTG = 1;

    if (this.faction == Game.PLAYER_FACTION) {
        multiplierTG = (int) Math.floor(numOfTrainingGrounds * 0.5);
    }

    basehp = basehp + 3*improveRatio *multiplierTG;
    hp = basehp;
    atk += (int) 3*improveRatio *multiplierTG;
    def += (int) 3*improveRatio*multiplierTG;

}
```

- The unit will be placed on the specified x and y positions, and its own stats will be updated.
- The unit is then placed on the player or enemy lists based on its faction.

The addBuilding method works in almost the exact same way (minus faction checks, as buildings can only be placed by the player.)

updateUnit and updateBuilding:

```
static public void updateUnit(Unit unit, ArrayList<Unit> u) {
    for (Unit i: u) {
        if (i.id == unit.id) {
            i.hp = unit.hp;
        }
    }
}
static public void updateBuilding(Batiment build, ArrayList<Batiment> u) {
    for (Batiment i: u) {
        if (i.id == build.id) {
            i.hp = build.hp;
        }
    }
}
```

These two methods are used whenever a unit or building take damage. They'll update the building and unit lists so that the elements on the lists can be equivalent to the elements on the map.

playerMoveUnit(Unit u)

```
static boolean playerMoveUnit(Unit u) {
    if(u.spd == 0) {
        System.out.println("This unit cannot move.");
        return false;
    }
    System.out.println("In what direction? (Up, Down, left, right)");
    String direction = sc.nextLine();
    int rangeChosen = 1;
    if(u.spd>1) {
        do {
            System.out.println("By how many squares? (value can go between 1 and " + u.spd + ").");
            rangeChosen = scanInt();
        } while (rangeChosen < 1 || rangeChosen > u.spd);
    }
    switch(direction.toLowerCase()) {
    case("u"):
    case("up"):
        return u.moveTo(0, -rangeChosen);
    case("d"):
    case("down"):
        return u.moveTo(0, rangeChosen);
    case("l"):
    case("left"):
        return u.moveTo(-rangeChosen, 0);
    case("r"):
    case("right"):
        return u.moveTo(rangeChosen, 0);
    default:
        System.out.println("Invalid direction.");
        return false;
    }

}
```

This method will move a player units depending on the direction a player inputs. It will also let the player choose how much a unit should move if it has a speed higher than 1.

attackUnits(Unit u)

```
static public boolean attackUnits(Unit u) {// if there is a unit to attack, attack it
    ArrayList<Unit> pT = new ArrayList<Unit>();// list that will get filled up will all units around attacking unit
    u.checkForUnits(pT);
    if (pT.size() == 0) {//no unit was found
        if(u.faction == PLAYER_FACTION)
            System.out.println("Couldn't find units to attack.");
        return false;
        }
```

This method checks if a unit u can attack around itself, and makes it do that based on a few parameters. Let's deconstruct it:

A list of units called "Potential Targets" is created, this is where all the targets of the unit who wishes to attack will be stored. To actually store these units, we run a unit method called checkForUnits(pT):

```
public void checkForUnits(ArrayList<Unit> potentialTargets){//Checks around unit if there are opposing units to attack
    int leftmost = x - range;
    int rightmost = x + range;
    int upmost = y - range;
    int downmost = y + range;
    //this is the attack square around the unit. It should attack any enemy who is in that square
    //if range = 1, it will check in a 3 by 3 square around the unit
    //if range = 2, it will check in a 5 by 5 square around the unit

    for (int i = upmost; i<=downmost; i++) {
        for (int j = leftmost; j<=rightmost; j++) {//Checks around the unit for enemies
            if (!Map.getTileFromCenter(j, i).isEmpty() && Map.getTileFromCenter(j, i).getElement().elementType == "Unit") {//if tile
                if(Map.getTileFromCenter(j, i).faction != this.faction) {//If the found unit's faction is different than the attacke
                    potentialTargets.add((Unit)Map.getTileFromCenter(j, i).getElement());// add to potential targets
                }//we use (Unit) to convert from the superclass Elements to the subclass Unit. It is safe to do so as we made sure t
            }
        }
    }
}
```

checkForUnits iterates through a square around a unit, and fills up the potentialTargets list with units of opposing faction to the attacking unit. Note how we have to use (Unit) to cast the Element found into an object of type Unit.

Once the potentialTargets list has been filled and we've determined that it isn't empty, a few scenarios will present themselves:

- If the list only contains one element, simply attack the element in that list directly.
- If it contains more than one unit to attack, two other scenarios appear:
    - If the attacking unit is a player unit, the player is prompted to choose which enemy unit he wishes to attack by typing its id.
    - If the attacking unit is an enemy unit, a method called lowestHP(pT) will return the player unit with the lowest hp around the enemy unit. It will then proceed to attack.

```
if (pT.size() == 1) { //check if there's only 1 unit to attack
    u2 = pT.getFirst();
    } else { //more than one
        if(u.faction == PLAYER_FACTION) {//player gets to choose who to attack
            do {
                System.out.println("Type out the id of the enemy unit you wish to attack. (type -1 to cancel the attack, this will not end y
                id = scanInt();
                sc.nextLine();
                if (id == -1) {
                    System.out.println("Attack has been cancelled.");
                    return false;
                }
                u2 = findUnitThroughID(id, pT);
                if (u2 == null)
                    System.out.println("No enemies around your unit with such an id.");
            } while(u2 == null);

        } else // enemy has to choose who to attack
            u2 = lowestHP(pT);// get unit with lowest hp
    }
```

Here is the lowestHp method, it simply iterates through the potentialTarget list to find the unit with the lowest hp. It then returns that unit.

```
static Unit lowestHP(ArrayList<Unit> pT) {//returns unit
    Unit lowestHP = pT.getFirst();
    for (Unit i: pT) {
        if (lowestHP.hp>i.hp)
            lowestHP = i;
    }
    return lowestHP;
}
```

The attack will then proceed as normal, using the Attack method found in the Units class.

```
    int damage = u.Attacking(u2);
    if (damage>0) {
        System.out.println(u.icon + " dealt " +RED+ damage + RESET+ " damage to " + u2.icon + "!");
        wait(100);
    }
    return true;
}
```

Note the “wait(x)” method after the attack was done. This is a custom method that poses the current thread by x ms, so as to make some notifications easier to grasp.

[This was found under this Stack Overflow post.]

```
public static void wait(int ms) {//function to wait in ms
    try
    {
        Thread.sleep(ms);
    }
    catch(InterruptedException ex)
    {
        Thread.currentThread().interrupt();
        System.out.println("Thread has been interrupted");
    }
}
```

spawnEnemy()

```
static void spawnEnemy() {
    switch(gameLevel) {
    case 1:
        Game.addUnit("Soldier", -2, -Map.visibleGrid, ENEMY_FACTION);
        break;
    case 2:
        Game.addUnit("Soldier", -3, -Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Archer", 3, -Map.visibleGrid, ENEMY_FACTION);
        break;
    case 3:
        Game.addUnit("Soldier", 0, Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Eagle", -2, -Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Soldier", 2, -Map.visibleGrid, ENEMY_FACTION);

        break;
    case 4:
        Game.addUnit("Soldier", -3, -Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Archer", 2, Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Soldier", 2, -Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Soldier", 0, Map.visibleGrid, ENEMY_FACTION);
        break;
    case 5:
        Game.addUnit("Archer",Map.visibleGrid , -4, ENEMY_FACTION);
        Game.addUnit("Soldier", Map.visibleGrid,-2, ENEMY_FACTION);
        Game.addUnit("Soldier",-Map.visibleGrid , 3, ENEMY_FACTION);
        Game.addUnit("Soldier", 1, -Map.visibleGrid, ENEMY_FACTION);
        Game.addUnit("Archer", -1, -Map.visibleGrid, ENEMY_FACTION);
        break;
    case 6:
        Game.addUnit("Archer",4, Map.visibleGrid , ENEMY_FACTION);
        Game.addUnit("Soldier",Map.visibleGrid-1 , Map.visibleGrid , ENEMY_FACTION);
```

A simple method that will spawn enemies around the map based on the current level. Called when turnsBeforeFight reaches 0.

levelIncrease()

```
static void levelIncrease() {
    gameLevel++;
    System.out.println("You've reached level " + gameLevel + "!" );
    if (gameLevel == endLevel) {
        System.out.println("You've conquered the enemy! Bask in golry as your kingdom remains undefeated for the rest of times!");
        Main.gameOver = true;
    } else {
        int numOfTrainingGrounds = 0;
        for (Batiment i: gameBuildings) {
            if (i.type == "Training Ground" && i.buildTime <= 0)
                numOfTrainingGrounds++;
        }
        for (Unit i: playerUnits) {
            i.train(numOfTrainingGrounds);
            i.hp = i.basehp;
            if (numOfTrainingGrounds > 0) {
                System.out.println(i.icon + " has trained! here are their stats: ");
                i.showStats();
                wait(100);
            }
        }
        if (gameLevel <= 8 && ( gameLevel != 4 )) {
            Map.visibleGrid++;
            System.out.println("Your influence grows, and with it, so does your territory:");
        }
    }
}
```

Called after all enemies have been killed, this method will:

- Increase the current level.
- End the game if we reach the endLevel (level 8 as of writing this)
- If the game hasn't ended, it will make units train: As in, their stats will increase based on the amount of Training Grounds on the map.
- Increase the visible area on specific levels.

turnAction()

```
static public void turnActions(){
    boolean playerElementDiedDuringThatTurn = false;//will be set to true if something if a buil
    playerElementDiedDuringThatTurn = deathBatiment(gameBuildings);//removes any buildings with

    if(!Main.gameOver) {//only executes rest of commands if command center is still alive
        if (playerElementDiedDuringThatTurn == false) {//makes sure the boolean if set to false
            playerElementDiedDuringThatTurn = deathUnit(playerUnits);
        } else {
            deathUnit(playerUnits);
        }
        deathUnit(enemyUnits);

        if(!enemyUnits.isEmpty() && playerElementDiedDuringThatTurn == false)//otherwise enemy w
            enemyMovement(enemyUnits.getFirst(), enemyUnits.getFirst().spd);


        if(enemyUnits.isEmpty()) {//only generate ressources if no enemies
            HashMap<String, Integer> oldRessources= new HashMap<>();//to check whether resources
            oldRessources.putAll(ressources.currentRessources);
            for(Batiment i: gameBuildings) {    //if building is built and a fight isn't happeni
                if (i.buildTime>0) {
                    System.out.println(RED + i.type + " is still under construction..."+RESET);
                    i.buildTime--;
                }else {
                    i.function(10);
                }
            }

            for (String i : ressources.currentRessources.keySet()) {//print resource value if it
                if (ressources.currentRessources.get(i) > oldRessources.get(i)) {
                    int amount = ressources.currentRessources.get(i) - oldRessources.get(i);
                    System.out.println(YELLOW+"Generated " + amount + " " + i + "!"+RESET);
                }
            }
            oldRessources.clear();
```

This method is called after a player's turn has ended. It does a number of things:

- It first calls the deathBatiment and deathPlayer/deathEnemy methods, making sure that any element with 0 hp is removed from the board by going through lists. (Note how the rest of turnActions after deathBatiment is only called if the game hasn't ended yet.)

```
static boolean deathUnit(ArrayList<Unit> u) {
    boolean unitDied = false;
    for(int i = u.size() - 1; i>= 0; i--) {
        if(u.get(i).hp <= 0) {
            Map.getTileFromCenter(u.get(i).x, u.get(i).y).removeElement();
            Map.getMap();
            System.out.print(RESET + "\n");
            System.out.print(u.get(i).icon + " has perished." );
            wait(100);
            u.remove(i);
            unitDied = true;
        }
    }
    return unitDied;
}
```

(Note how the deathBatiment method also checks if the building that got destroyed is the command center. In that case, gameOver will be set to true and the game will end immediately.)

```
tic boolean deathBatiment(ArrayList<Batiment> b) {
 boolean buildingDied = false;
 for(int i = b.size() - 1; i>= 0; i--) {
     if(b.get(i).hp <= 0) {
         Map.getTileFromCenter(b.get(i).x, b.get(i).y).removeElement();
         if (b.get(i).type == "Command Center") {//if the destroyed building was the
             Main.gameOver = true;
             Map.getMap();
             //System.out.println(RED + "THE COMMAND CENTER HAS BEEN DESTROYED" + RES
             System.out.print("\n" + RED);
             for (char c : "THE COMMAND CENTER HAS BEEN DESTROYED".toCharArray()) {
                 System.out.print(c);
                 wait(200);
             }

             wait(2000);
             System.out.print("\n" + RESET);
             break;
         }
         System.out.print(b.get(i).icon + " has been destroyed." );
         wait(100);
         gameBuildings.remove(i);
         buildingDied = true;
     }
 }
 return buildingDied;
}
```

- Adding to that, if a player unit or a building was destroyed during that turn, the playerElementHasDiedDuringThatTurn variable will be set to true. This is to make sure an enemy won't move immediately after a building or player unit has been eliminated.
- The game will then check if there are any enemies on the board: In that case, and if a building or player hasn't been eliminated beforehand, the enemyMovement method will be called. More on that in a bit. For now, note that the enemy chosen to move was the first one on the enemyUnits list.
- After which comes resource generation: If there are currently no enemies on the map, the game first saves the current resources in a hashmap called oldResources. It then goes through all the buildings in the gameBuildings list and makes them carry their functions (i.e generate ressources). It will then print all the resources that changed during this round by comparing all of them with what can be found in the oldResources hashmap.

enemyMovement(Unit u, int speed)

Perhaps the most complicated method, this will move an enemy unit based on a specific ruleset:

- The enemy will not move if there are any player units around it: It will prioritise defending itself instead by attacking.
- If there are no players around it, two scenarios present themselves:
    - If the enemy has an attack range higher than 1, it will try and attack the command center. It will perform its normal movement logic if it fails to find it.
    - If the enemy has a range of 1, it will perform its normal movement logic.
- The enemy's movement logic works like this:
    - If the command center is below me, my destination will be a tile under me. The tile I travel to will be based on my speed, unless it means I'll overshoot. In that case, I will travel to the tile before it.
    - If the command center is instead above me, I'll follow a similar logic.
    - if the command center is neither below nor above me, then it must mean I'm on the same row as it. In that case:
        - If the command center is to my right, I'll go to the right, with a logic identical to the one described beforehand.
        - If the command center is not on my right, then it must be on my left, in that case, I'll go to the left, with that same logic.
- Before proceeding, the enemy has another check to make if the tile it travels to is not right next to it: (ie, it can travel more than one tile):
    - If the tile I travel to isn't empty, then I'll check the one before it until I can only travel to the tile right next to me.

- After the enemy has decided of its destination tile, it will analyse its contents and act according to this logic:
    - If I'm travelling to an empty tile, I will move there.
    - If it's not empty, I must assume it's the tile right next to me, in that case:
        - If I'm travelling to a tile and it contains a unit of my faction, then that unit will have to move using a similar logic to mine. (recursivity)
        - If that tile contains a building, then that means that this building is blocking my path and that I cannot go over it. In that case, I'll attack it.

With this logic system, all enemies are sure to reach the Command Center, even if it's not the optimal path. We might replace this with a more straightforward pathfinding system in the future once we have more experience, but this is how it works for now.

```
static void enemyMovement(Unit u, int speed) {
    if (!attackUnits(u)) {//only move if no opposing units around you
        boolean hasAttacked = false;
        if (u.range != 1) { // if unit has a range above 1 square, try to find command center
            if (Math.abs(u.x) - u.range <= 0 && Math.abs(u.y) - u.range <= 0) {//if centre de commande in range, attack it
                u.Attacking((Batiment)Map.getTileFromCenter(0, 0).getElement());
                hasAttacked = true;// won't move if it attacked command center
            }
        }
```

```
if (!hasAttacked) {
    int movX=0, movY=0;// by how much unit will move in x xor y
    if (u.y<0) {// check if command center below you
        for(int i = speed; i>0; i--) {
            if (u.y+ i <= 0) {
                movY = i;
                break;
                }
            }
        }
    else if (u.y>0) {
        for(int i = speed; i>0; i--) {
            if (u.y- i >= 0) {
                movY = -i;
                break;
                }
            }
        }
    else if (u.x <0) {
        for(int i = speed; i>0; i--) {
            if (u.x+ i <= 0) {
                movX = i;
                break;
                }
            }
        }
    else {
        for(int i = speed; i>0; i--) {
            if (u.x- i >= 0) {
                movX = -i;
                break;
                }
            }
        }
```

```
while ( (movX < -1 || movX > 1) && !Map.getTileFromCenter(u.x + movX, u.y + movY).isEmpty()) {
    int dec =(int) -Math.signum(movX);
    movX += dec;

}



while ( (movY < -1 || movY > 1) && !Map.getTileFromCenter(u.x + movX, u.y + movY).isEmpty()) {
    int dec =(int) -Math.signum(movY);
    movY += dec;
}

///////

if (!Map.getTileFromCenter(u.x + movX, u.y + movY).isEmpty()) {//if the tile we wish to move to isn
    if (Map.getTileFromCenter(u.x + movX, u.y + movY).getElement().elementType == "Building") {
        u.Attacking((Batiment)Map.getTileFromCenter(u.x+movX, u.y + movY).getElement()); // we know
    } else {
        Unit u2 = (Unit)Map.getTileFromCenter(u.x+movX, u.y + movY).getElement();
        if (u2.faction == ENEMY_FACTION) {
            enemyMovement(u2, u2.spd);
        } else {
            enemyMovement(u, u.spd - 1);
        }

    }
} else {// tile is empty
    u.moveTo(movX, movY);
}
```

Sources:

- The W3Schools website: For learning how to implement concepts such as collections and try / catch errors, which can be a little tricky
- Patorjk: Tool used to generate the logo of the game with ASCII Art
- Stack Overflow: A very useful site for finding solutions to specific issues. Most notably, this is where the CustomColors code was found.

Things we wish to improve on regarding this project:

- Implementing a more robust pathfinding algorithm for enemies.
- Adding more units and building types.
- Adding more levels and enemy waves.
- Rethinking the map system to possibly increase performance.

Conclusion:

This project has enabled us to expand our knowledge on Object-Oriented Programming, as well as work on coding with Java using its key concepts: Classes, Object, Encapsulation, Inheritance, Polymorphism, Abstraction and Collections.

We are certain that with the experience acquired whilst making this project, and the new information that will be gained throughout the next few years, the game's performance and overall quality will increase to a favorable level.

Thank you for your time.