

Lesson 9 - Team assignment

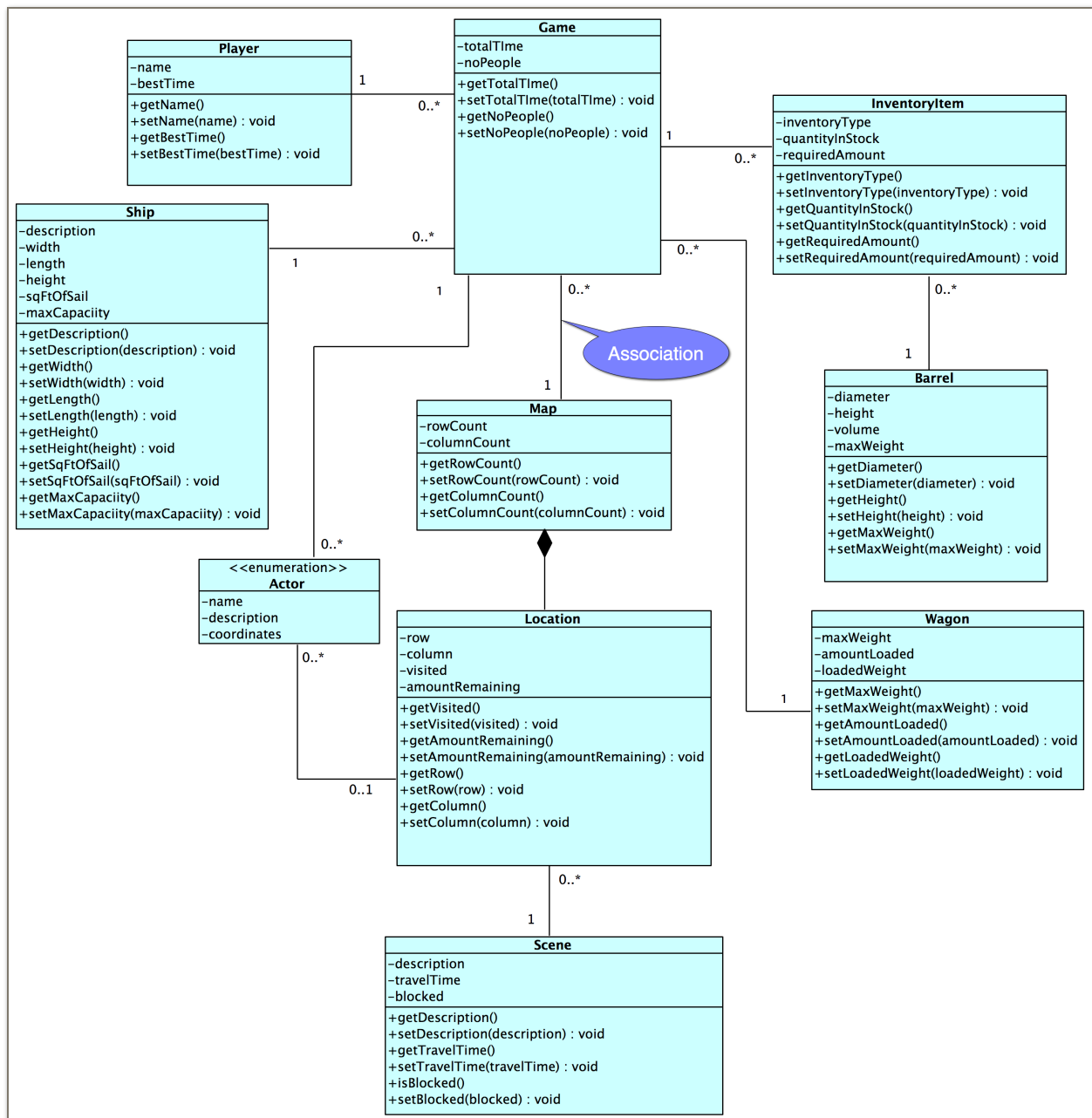
In this team assignment you will be working with the `array`, `ArrayList` and `enums` data types. First, we will start by learning how to implement the association relationships between the Model Layer classes in the UML Model Layer Class diagram. This is done by adding variables to each of the Model Layer classes for each end of the relationships. Some of these variables will have to model a list of objects whose data type is either an `array`, `ArrayList` or `enum`.

Once these relationship and objects are defined you will then implement the “Start new game” option in the `MainMenuView` class of your program. This will require that you create a new `Game` object and all of the objects associated with the `Game` object in your UML Model Layer class diagram. Many of these objects will be `arrays`, `ArrayLists` and `enums`.

Implementing association relationships

Here is a UML Class Diagram for the Model Layer of the example game showing all of the associations to the `Game` class. We created Java bean for each of the Model Layer classes earlier. For each of these, we implemented the `Serializable` interface, created a default constructor function and made all if the class instance variables private with public “getter and setter” functions. We did not, however, implement any of the association relationships between the classes. The reason for this is that most of the associations have a cardinality of one-to-many (`1..*`) or many-to-many (`*..*`). Java `arrays`, `ArrayList` or `enums` are required to implement these type of relationships.

Before you move on you may need to go back and update the associations in your Model Layer class diagram to have the correct cardinality for each association. If the associations are wrong then your implementation will be incorrect and your program will not work correctly. Use the UML class diagram below as a guide to what the associations should look like in your program.

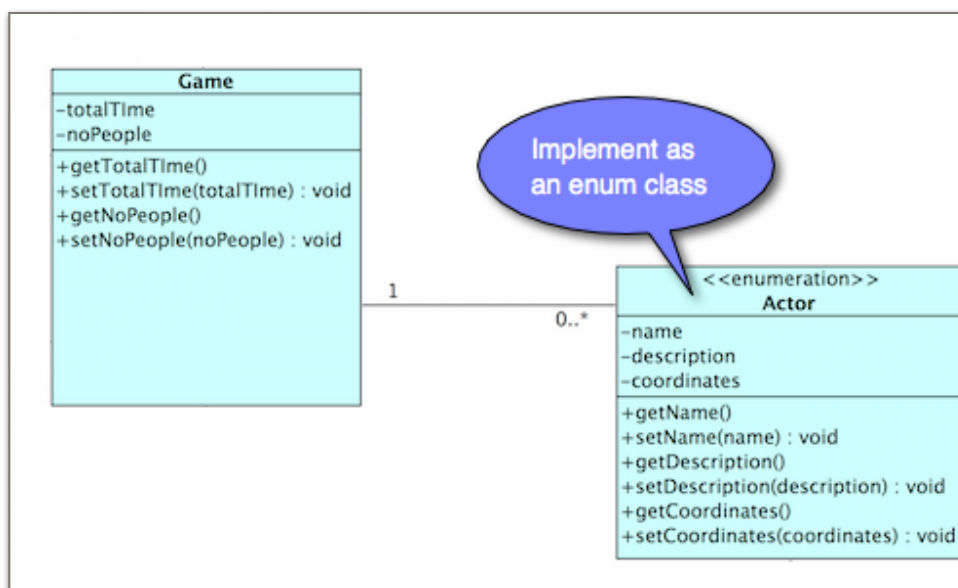


Step 1 - Pull from the repository

Pull the latest changes from the your GitHub repository.

Step 2 - Modify the Actor class to be the Actor enum

Notice in the class diagram that the `Actor` class is defined as an enumeration. Before we can implement the association between the `Game` and `Actor` (or `Character`) classes we need to change the `Actor` class into an `enum` class. We chose to make the `Actor` class and `enum` because the list of actors in the game is constant and will never change.



We start by modifying the regular `Actor` class that we created when we implemented each of the Model Layer classes as Java Beans.

The first thing to do is modify the `class` keyword to the `enum` keyword in the class definition statement at the top of the class.


```
public class Actor implements Serializable {  
    ...  
}  
public enum Actor implements Serializable {  
    ...  
}
```

An orange arrow points from the `class` keyword in the first line to the `enum` keyword in the second line.

Several errors will occur in your class. Ignore these areas for now. Many of the errors will go away once you have made the following changes.

The keyword for element in the enum list will be the name of the actor so we need to delete the `name` instance variable along with its “getter” and “setter” methods (i.e., `getName()` and `setName()`).

```
public class Actor implements Serializable {  
    private String name;  
    private String description;  
    private Point coordinates;  
  
    public Actor() {  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```




Next we need to add the `final` keyword to all of the private instance variables because the values of an enum class are constant and can not be changed.

```
private final String description;  
private final Point coordinates;
```


Now we need to modify the constructor function to be an overloaded constructor that will initialize each of the class instance variables with a value. The value of the `description` will be passed in as a parameter and assigned to the `description` instance variable. A new `Point` object is then created and initialized to the starting position of the actor in the map. The `Point` object is then assigned to the `coordinates` instance variable.

```
public Actor() {  
}  
  
Actor(String description) {  
    this.description = description;  
    coordinates = new Point(1,1);  
}
```




We also now need to delete all of the “setter” functions for the class attributes because the values of all of the class attribute variables are constant and cannot be changed in an enum class. In the example program, we deleted the “setter” functions for the `description` and `coordinates` class attribute variables.

```
public String getDescription() {  
    return description;  
}  
  
public void setDescription(String description) {  
    this.description = description;  
}  
  
public Point getCoordinates() {  
    return coordinates;  
}  
  
public void setCoordinates(Point coordinates) {  
    this.coordinates = coordinates;  
}
```



Next, we need to delete the `hashCode()` and `equals()` functions from the enum class. These functions are used to compare one object instance with another and to sort search for items in the list quickly. Neither of these functions is meaningful for an enum class. No two objects in an enum list will ever be equivalent and we never need to search the enum list because each item is referenced by it's enum keyword name.

```
@Override  
public int hashCode() {  
    int hash = 7;  
    return hash;  
}  
  
@Override  
public boolean equals(Object obj) {  
    if (obj == null) {  
        return false;  
    }  
    if (getClass() != obj.getClass()) {  
        return false;  
    }  
    final Actor other = (Actor) obj;  
    if (!Objects.equals(this.description, other.description)) {  
        return false;  
    }  
    return true;  
}
```



Finally we need to define the each of the elements in the enum list. This is done by defining the list of keywords and passing the value of the description to the new overloaded constructor function just created.

```
public enum Actor implements Serializable {  
  
    Lehi("He is the prophet and leader of the family."),  
    Sariah("She is Lehi's wife and mother of the family."),  
    Nephi("Faithful son and later the prophet leader of the Nephites."),  
    Jacob("Nephi's faithful brother, prophet and successor to Nephi"),  
    Sam("The youngest boy and faithful brother of Nephi."),  
    Laman("The oldest rebellious brother and leader of the Lamanites."),  
    Lemuel("The 2nd oldest rebellious brother who went with Laman"),  
    Zoram("Laban servant that became a faithful follower of Nephi");  
  
    private final String description;  
    private final Point coordinates;  
  
    Actor(String description) {  
        this.description = description;  
        coordinates = new Point(1,1);  
    }  
  
    public String getDescription() {  
        return description;  
    }  
  
    public Point getCoordinates() {  
        return coordinates;  
    }  
  
}
```

Instructions:

Create the an enum class for the list of actors (characters) in your game. Then create enum list for the other fixed list of items in your game.

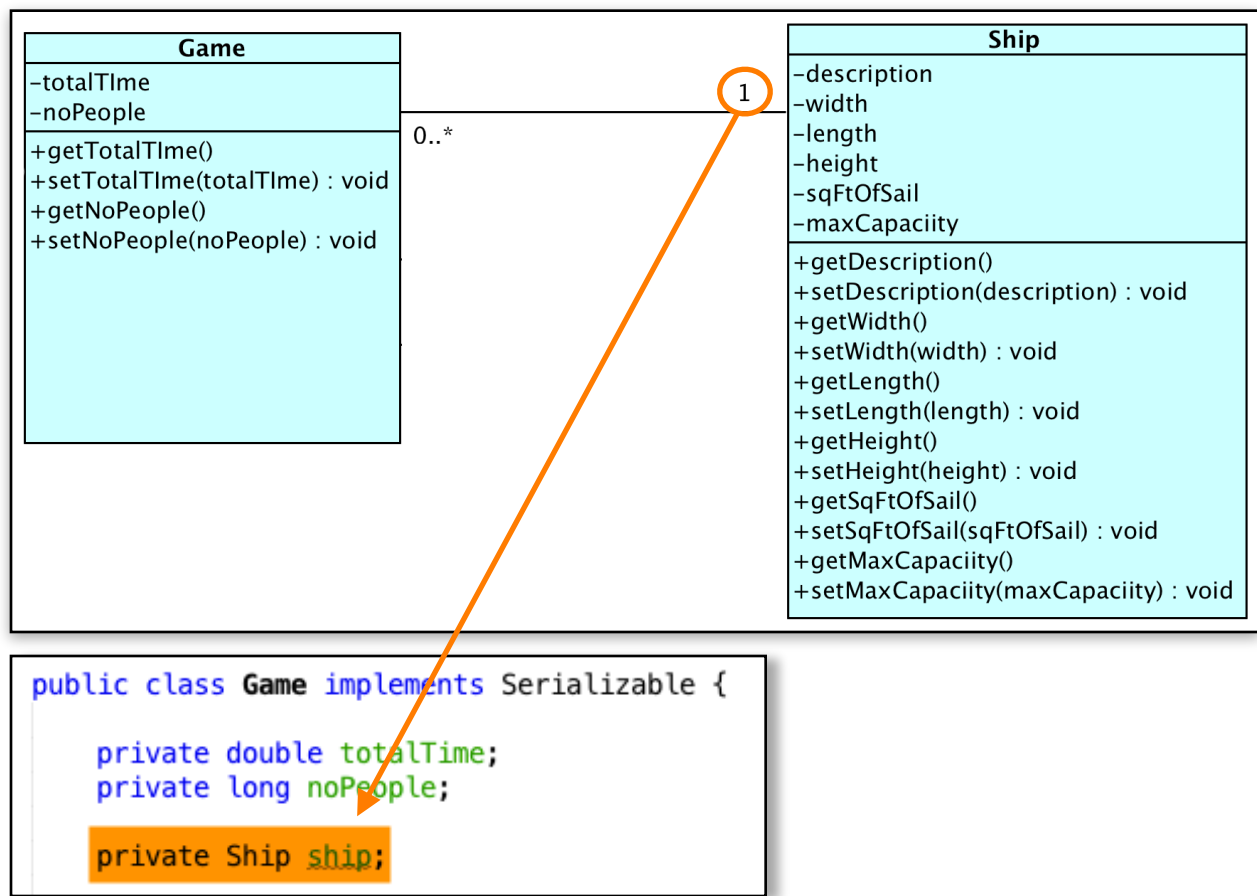
Step 3 - Implement the Model Layer Associations

Implement each of the associations between the Model Layer classes in your UML class diagram. Do not worry about the dependency relationships to the Control Layer classes and the relationships to the View Layer classes for now.

The associations in the UML class diagram are implemented by adding class instance variables to the each Model Layer class for the cardinality on both ends of each association, aggregation and composition relationship.

Implementing a cardinality of one

If the cardinality on the one end of the association relationship in the UML Class diagram is a 0..1 or 1 then we add a class instance variable to the class on the opposite end of the association variable whose data type is that of the class defined on the side of the relationship where the cardinality of one is specified.



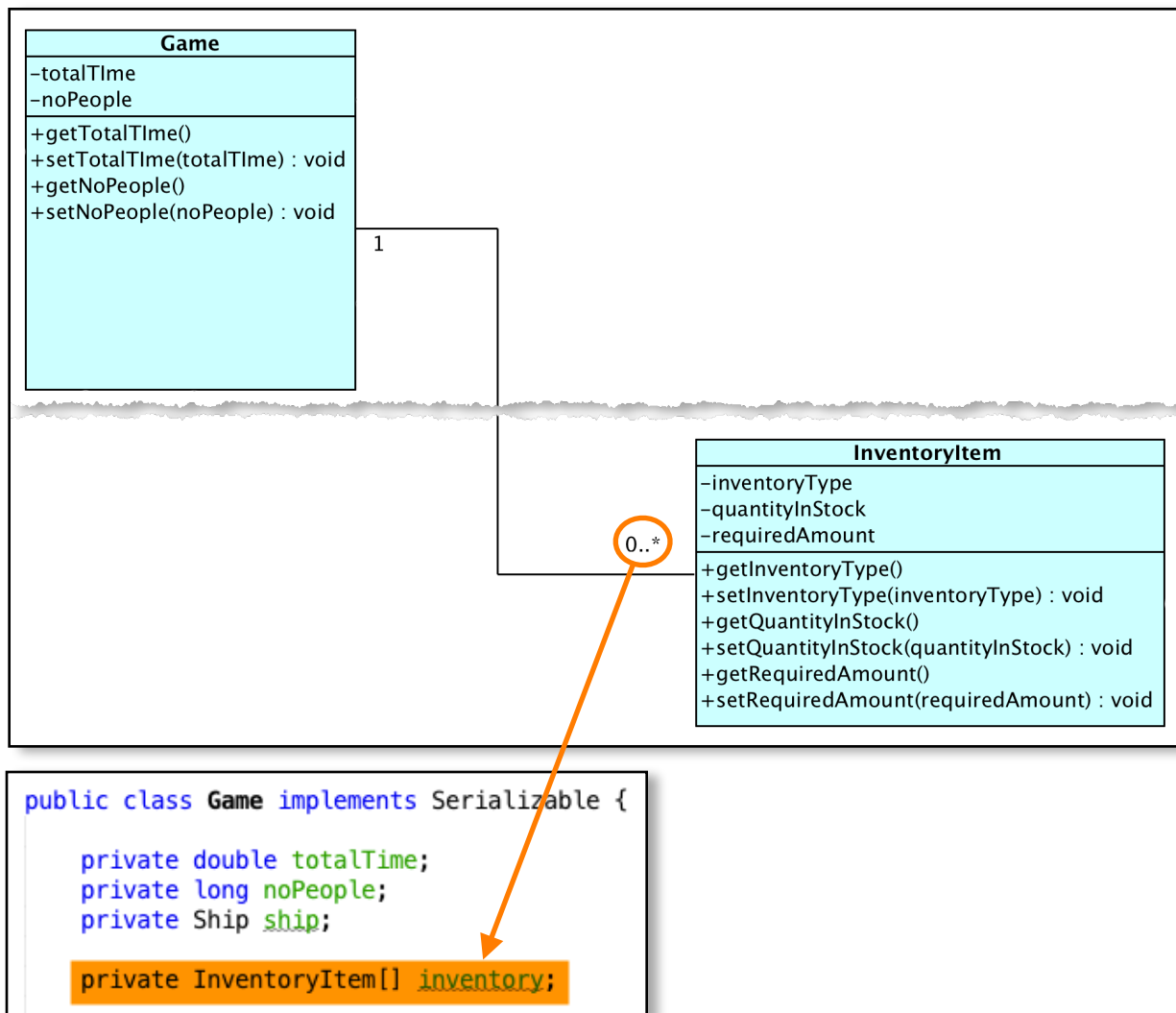
For example, the Game-Ship association above the Game class has one and only one Ship. A class instance variable named ship of the Ship data type is added to the Game class on the opposite side of the relationship. This variable will contain a reference to the ship being used in the game after we have assigned a Ship object to it.

Add a “getter” and “setter” function for the new class variable you just defined. Move your cursor down to the end of the existing list of the “getter” and “setter”. Use the **Insert code** menu command to create the “getter” and “setter” functions.

```
public long getNoOfPeople() {  
    return noOfPeople;  
}  
  
public void setNoOfPeople(long noOfPeople) {  
    this.noOfPeople = noOfPeople;  
}  
  
public Player getPlayer() {  
    return player;  
}  
  
public void setPlayer(Player player) {  
    this.player = player;  
}
```


Implementing a cardinality of many

When the cardinality on the other end of the association relationship is $0..*$, or $1..*$, an instance variable is added to the class on the opposite side of the relationship whose data type must be either `array` or an `ArrayList`. We choose the `array` data type if the number of items in the list is fixed. If the number of items in the list may vary during execution, we choose the `ArrayList` data type.



The `Game-InventoryItem` association below indicates that a game has zero-to-many ($0..*$) `InventoryItem` objects and each `InventoryItem` must have one and only one (1) game. We added a `inventory` class instance variable to the implementation of the `Game` class. Its data type is an array (i.e., list) of `InventoryItem` objects. We chose the array data

type because in the example game there is a fixed number of inventory items in the inventory list. The `inventory` variable in the `Game` class references the array (list) of `InventoryItems` in the game.

We then added class instance variables for all of the remaining associations linked to the `Game` (e.g., `Player`, `Wagon`, `Actors`, and `Map`). After defining the class instance variables for each of the associations connected to the `Game` class, we created “getter” and “setter” functions for each of the new instance variables.

```
public class Game implements Serializable {  
  
    private double totalTime;  
    private Ship ship;  
    private InventoryItem[] inventory;  
    private Player player;  
    private String[] actors;  
    private Wagon wagon;  
    private Map map;  
  
    public Game() {  
    }  
  
    public double getTotalTime() {  
        return totalTime;  
    }  
  
    public void setTotalTime(double totalTime) {  
        this.totalTime = totalTime;  
    }  
  
    public Ship getShip() {  
        return ship;  
    }  
  
    public void setShip(Ship ship) {  
        this.ship = ship;  
    }  
  
    public InventoryItem[] getInventory() {  
        return inventory;  
    }  
}
```

Instructions:

Create the class instance variables for each of the associations relationships connected to the `Game` class in your modified Model Layer UML Class diagram for your program. If the cardinality is (0..1) or (1), you will add an variable who data type is a single object. If the cardinality is (0..*) or (*..*) you will need to define a variable that is of the array, `ArrayList` or `enum` data type. Choose an `array` if the number of items in the the list will not change during the game. For example the length of the list of inventory items associated with the `Game` object will never change. Choose an `ArrayList` if the number items in the list may vary during the game. For example, the length of the list of games associated with the player will vary during the execution of the program. Choose an `enum` if the number of items in the list will never change and the objects in the list are constant (i.e., their values can not change). For example, the list of Actors will be constant through out the game so we will implement this as an `enum`.

Then create the “getter” and “setter” functions for each of these new class instance variables.

Implementing the functions to create a new game

The next step in developing our program is to implement the the `doAction()` function in the `MainMenuView` class associated with the Start new game command in the menu. This function is responsible for calling the Control Layer function to actually create a new game and start the play of the game. Then it will display the game view.

```
Main Menu:
```

```
N - Start new game
G - Start saved game
H - Get help on how to play the game
S - Save game
Q - Quit
```

You should have created a `doAction()` function and the appropriate stub functions in the `MainMenuView` class for each of the menu items when you first implemented the `MainMenuView` class. The `doAction()`

```

public void doAction(char choice) {
    switch (choice) {
        case 'N': // create and start a new game
            this.startNewGame();
            break;
        case 'G': // get and start an existing game
            this.startExistingGame();
            break;
        case 'H': // display the help menu
            this.displayHelpMenu();
            break;
        case 'S': // save the current game
            this.saveGame();
            break;
        case 'E': // Exit the program
            return;
        default:
            System.out.println("\n*** Invalid selection *** Try again");
            break;
    }
}

```

function should be calling the stub function associated with the corresponding menu item as shown below.

```

62 private void startNewGame() {
63     // create a new game
64     GameController.createNewGame(CuriousWorkmanship.getPlayer());
65
66     // display the game menu
67     GameMenuView gameMenu = new GameMenuView();
68     gameMenu.display();
69 }

```

You now are going to implement the `startNewGame()` stub function for the Start new game menu item. This function needs to call a Control Layer function to create a new game and then display the Game Menu view to start the game.

Here is the implementation of the `startNewGame()` function in the `MainMenuView` class in the example program.

```

public class GameController {

    public static void createNewGame(Player player) {
        System.out.println("*** createNewGame in GameController called ***");
    }
}

```

The first statement calls a Control Layer function called `createNewGame()` to be implemented in the `GameControl` class. The `Player` object saved in the `CuriousWorkmanship` class when we first started the program is passed to the function. A new `GameMenuView` object instance is then created and its `displayMenu()` function is called to display the game menu.

We then created the `createNewGame()` stub function in the `GameControl` class.

Instructions:

Create the `startNewGame()` function and a stub for the `createNewGame()` function for your game.

Implement the `createNewGame()` Control Layer function

The `createNewGame()` function is then implemented. This function is responsible for creating a new game and then moving the actors to the starting location in the map.

The `Game` class has many different objects associated with it. We need to assign values to each of the class instance variables added for the associations when we create a new `Game` object.

```
public class Game implements Serializable {  
    private double totalTime;  
    private Player player;  
    private InventoryItem[] inventory;  
    private Wagon wagon;  
    private Ship ship;  
    private Map map;  
}
```

The `Player` object was created earlier and saved in the `ProgramControl` class when the program is first started program and the users is prompted to enter their name. The other associated objects (inventory, wagon, ship and the map) have not be created yet. We need

to create each of these objects before we can save them in the new Game object.

Here is the algorithm for the `createNewGame()` function in the example program.

```
createNewGame(): void
BEGIN
    create a new game
    save the game in CuriousWorkmanship

    get player from CuriousWorkmanship
    save player in the game

    create list of inventory items
    save inventory list in game

    create the wagon
    save wagon in the game

    create the ship
    save ship in the game

    create the map
    save map in the game

    move actors to starting location
END
```

Here is the implementation of the `createNewGame()` function.

```
29 public class GameController {
30
31     public static void createNewGame(Player player) {
32
33         Game game = new Game(); // create new game
34         CuriousWorkmanship.setCurrentGame(game); // save in CuriousWorkmanship
35
36         game.setPlayer(player); // save player in game
37
38         // create the inventory list and save in the game
39         InventoryItem[] inventoryList = GameController.createInventoryList();
40         game.setInventory(inventoryList);
41
42         Ship ship = new Ship(); // create new ship
43         game.setShip(ship); // save ship in game
44
45         Wagon wagon = new Wagon(); // create new wagon
46         game.setWagon(wagon); // save wagon in game
47
48         Map map = MapControl.createMap(); // create and initialize new map
49         game.setMap(map); // save map in game
50
51         // move actors to starting position in the map
52         MapControl.moveActorsToStartingLocation(map);
53     }
```

We first started by creating a new instance of the `Game` class and then called the `setCurrentGame()` “setter” function in the `ProgramControl` class to save a reference to the current game. We then saved the player object passed to `player` parameter when the `createNewGame()` function is called. This is done by calling the `Game` class’s `setPlayer()` “setter” function. Next we used the “Divide and Conquer” technique to create separate functions to create the inventory items. Each of these functions will be implemented at a later time. These two lists are saved in the game by calling the `Game` object’s respective “setter” functions. We then create new `Wagon` and `Ship` objects and saved those in the game using the appropriate “setter” function. Lastly, we used “Divide and Conquer” again to create a function to create the `Map` object and to move the actors to the starting location in the map.

We then created stub functions for each of the new functions called in the appropriate Control Layer classes (i.e., the `createInventoryList()` stub functions in the `GameControl` class, and the `createMap()` and the `moveActorsToStartingLocation()` stub functions in the `MapControl` class).

```
public class GameControl {  
  
    public static void createNewGame(Player player) {...23 lines }  
  
    public static InventoryItem[] createInventoryList() {  
        System.out.println("*** called createInventoryList() in GameControl ***");  
        return null;  
    }  
}
```

```
public class MapControl {  
  
    public static Map createMap() {  
        // create the map  
        Map map = new Map(20, 20);  
  
        // create a list of the different scenes in the game  
        Scene[] scenes = createScenes();  
  
        // assign the different scenes to locations in the map  
        assignScenesToLocations(map, scenes);  
  
        return map;  
    }  
}
```

Instructions:

Implement the `createNewGame()` functions for your program.

Implement the `createInventoryList()` function

The stub function `createInventoryList()` is responsible for creating a list of `InventoryItem` objects that are to be associated with the game.

Here is the implementation of the `createActorList()` function in the example game. This function first creates a array of the `InventoryItem` objects in game.

```
public static InventoryItem[] createInventoryList() {  
    // created array(list) of inventory items  
    InventoryItem[] inventory =  
        new InventoryItem[14];  
  
    InventoryItem lumber = new InventoryItem();  
    lumber.setDescription("Lumber");  
    lumber.setQuantityInStock(0);  
    lumber.setRequiredAmount(0);  
    inventory[0] = lumber;  
  
    InventoryItem ore = new InventoryItem();  
    ore.setDescription("Ore");  
    ore.setQuantityInStock(0);  
    ore.setRequiredAmount(0);  
    inventory[1] = ore;  
  
    InventoryItem grain = new InventoryItem();  
    grain.setDescription("Grain");  
    grain.setQuantityInStock(0);  
    grain.setRequiredAmount(0);  
    inventory[2] = grain;  
  
    InventoryItem saw = new InventoryItem();  
    saw.setDescription("Saw");  
    saw.setQuantityInStock(0);  
    saw.setRequiredAmount(2);  
    inventory[12] = saw;  
  
    InventoryItem nails = new InventoryItem();  
    nails.setDescription("Sickle");  
    nails.setQuantityInStock(0);  
    nails.setRequiredAmount(50);  
    inventory[13] = nails;  
  
    return inventory;  
}
```

First, we defined an array of 14 `InventoryItem` objects and assigned it to the variable `inventory`. Then we created a new `InventoryItem` object for each item in `inventory` array and assigned it to the next position in the array. Finally, we returned the `inventory` array.

The `inventory` array will be quite difficult to use in the rest of the program because we will have to memorize the position of each `InventoryItem` object in the array to be able to access it. For example, to access the grain `InventoryItem` object we would need to hardcode the index the object in the array (i.e., `inventory[2]`) every place in our program that we need to access the grain item. This makes our code very brittle and cause errors if we ever move the item to a different location in the `inventory` array or we add or delete items in the `inventory` array.

We can easily solve this problem and make the code much more readable by creating an enum list for the index position of each `InventoryItem` object in the `inventory` array.

```
public enum Item {  
    lumber,  
    ore,  
    grain,  
    legume,  
    oil,  
    water,  
    honey,  
    salt,  
    axe,  
    hammer,  
    drill,  
    shovel,  
    sickle,  
    saw,  
    nails;  
}
```

We only defined only a list of keywords for each element in the `Item` enum list. Each keyword is automatically assigned an ordinal value starting with 0 (e.g., 0, 1, 2, 3, ... 12, 13). The ordinal values will correspond to the index position of each `InventoryItem` object in the `inventory` array. No attributes needed to be defined in the `Item` enum

list so we did not need to create an overloaded constructor function or “getter” functions.

We now modified the `createInventoryList()` function to use the `Item` enum list to access the individual items in the inventory array. For each item in the inventory list we get the index of the element by specifying the keyword defined for the item in the `Item` enum class followed by a call to its `ordinal()` function. The `ordinal()` function returns the ordinal number or the index of the enum assigned to the keyword.

```
public static InventoryItem[] createInventoryList() {  
    // created array(list) of inventory items  
    InventoryItem[] inventory =  
        new InventoryItem[Constants.NUMBER_OF_INVENTORY_ITEMS];  
  
    InventoryItem lumber = new InventoryItem();  
    lumber.setDescription("Lumber");  
    lumber.setQuantityInStock(0);  
    lumber.setRequiredAmount(0);  
    inventory[Item.lumber.ordinal()] = lumber;  
  
    InventoryItem ore = new InventoryItem();  
    ore.setDescription("Ore");  
    ore.setQuantityInStock(0);  
    ore.setRequiredAmount(0);  
    inventory[Item.ore.ordinal()] = ore;  
  
    InventoryItem grain = new InventoryItem();  
    grain.setDescription("Grain");  
    grain.setQuantityInStock(0);  
    grain.setRequiredAmount(0);  
    inventory[Item.grain.ordinal()] = grain;  
}
```

Anytime we want to access a particular item in the `inventory` array all we do is specify the corresponding keyword in the `Item` enum list. For example to access the grain `InventoryItem` object we would specify the following.

```
inventory[Item.grain.ordinal()]
```

We no longer need to remember the index of the grain item in the inventory list, and we do not care if we move it to a different position in the list. Furthermore the code is self documenting. It is clear from looking at the code which item in inventory is being accessed.

Instructions:

Create an enum list for the index positions of the inventory items in the inventory list for your game, and then create the `createInventoryList()` function for your program.

Override constructor functions for the Wagon and Ship classes

In the the `createNewGame()` function the `GameControl` class we called the default constructor functions to create the `Ship` and `Wagon` objects for the game.

```
public class GameControl {  
    public static void createNewGame(Player player) {  
        Game game = new Game(); // create new game  
        CuriousWorkmanship.setCurrentGame(game); // save in CuriousWorkmanship  
  
        game.setPlayer(player); // save player in game  
  
        // create the inventory list amd save in the game  
        InventoryItem[] inventoryList = GameControl.createInventoryList();  
        game.setInventory(inventoryList);  
  
        Ship ship = new Ship(); // create new ship  
        game.setShip(ship); // save ship in game  
  
        Wagon wagon = new Wagon(); // create new wagon  
        game.setWagon(wagon); // save wagon in game  
  
        Map map = MapControl.createMap(); // create and initialize new map  
        game.setMap(map); // save map in game  
  
        // move actors to starting position in the map  
        MapControl.moveActorsToStartingLocation(map);  
    }  
}
```

We need to override both of these functions to set the initial values of the attributes for both of these classes.

Here are the modifications to the default constructor for the `Ship` class.

```
public class Ship implements Serializable {  
  
    private String description;  
    private double width;  
    private double length;  
    private double height;  
    private double sqFtOfSale;  
    private double maxWeighCapacity;  
  
    public Ship() {  
        this.description = "\nThis is the this of curious workmanship. If you "  
            + "\nhave faith, it will take you to the promised land";  
        this.width = 0;  
        this.length = 0;  
        this.height = 0;  
        this.maxWeighCapacity = 0;  
        this.sqFtOfSale = 0;  
    }  
}
```

Here are the modifications to the default constructor for the `Wagon` class.

```
public class Wagon implements Serializable {  
  
    private long numberBarrelsLoaded;  
    private double maxWeight;  
    private double loadedWeight;  
  
    public Wagon() {  
        this.numberBarrelsLoaded = 0;  
        this.maxWeight = 1000;  
        this.loadedWeight = 0;  
    }  
}
```

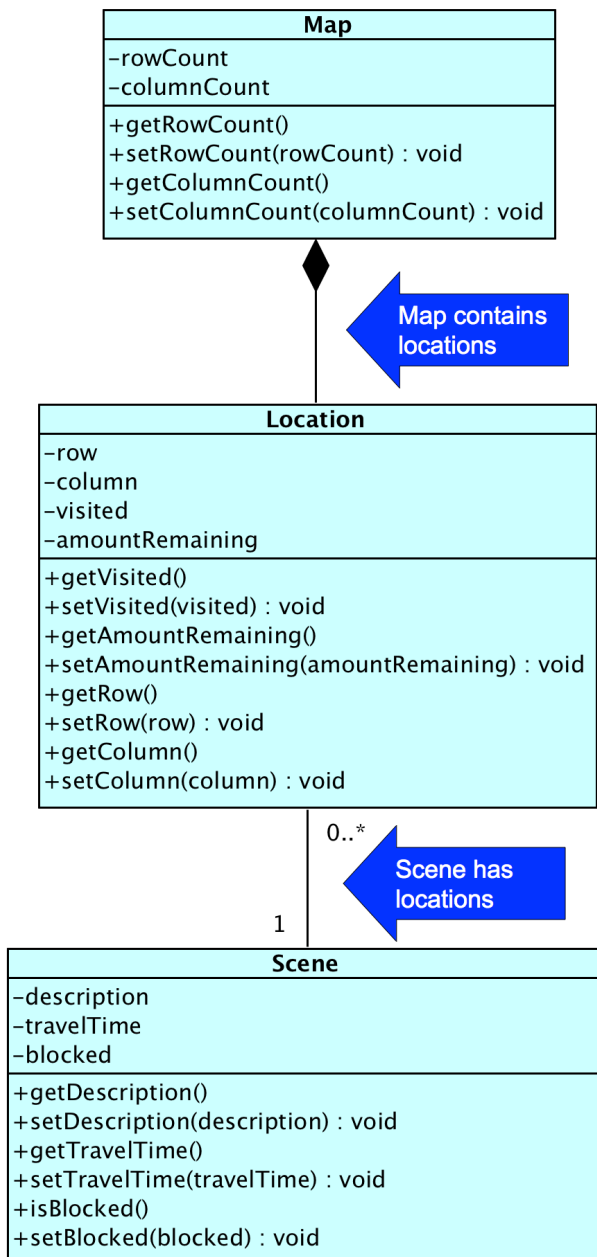
Instructions:

Implement any other classes and/or override the constructor functions needed for your game.

Implementing the `createMap()` function

The `createMap()` function is responsible for creating and populating the map for the game.

In the UML Class Diagram for the example game there is a composite relationship between a `Map` and `Location` (a `Map` “contains” many `Location` objects, and a one-to-many relationship between a `Scene` and a `Location`).



We added a class instance variables in the `Map` class to store a reference to a two dimensional array of the `Location` objects associated with current map. We then added “getter” and “setter” functions for the `locations` class instance variable to the `Map` class.

```
public class Map implements Serializable {  
    private int noOfRows;  
    private int noOfColumns;  
    private Location[][] locations;
```

There are also associations defined between the `Location` class and the `Scene` classes and the `Location` and `Actor` classes in the UML Class Diagram (i.e., a `Location` has one `Scene`). We added a class instance variable to the `Location` class to reference the `Scene` object assigned to the location. Lastly, there is an association between between `Location` and `Actor` classes (i.e., a `Location` may have one-to-many `Actors`). A class instance variable is also added to the class to reference the list of `Actor` objects assigned to this location. We then created “getter” and “setter” functions for the `scene` and `actors` class instance variables.

```
public class Location implements Serializable {  
    private int row;  
    private int column;  
    private boolean visited;  
    private Scene scene;  
    private ArrayList<Actor> actors;
```

Now we can define the algorithm for the `createMap()` function. First, we create the map for the game and all of the locations in the map. Then we create all of the scenes to be assigned to the different locations in the map and finally we assign the scenes to the various locations in the map.

```
createMap(): Map
BEGIN
    Create and initialize new map
    Create all of the scenes for the map
    Assign each scene to a location in the map
END
```

Here is the code for the `createMap()` function. Again, we used “Divide and Conquer” to create and call functions for each of the three subtask defined in the algorithm. We first call an overloaded constructor function to create the `Map` object and all of the `Location` objects in the map. Then we call the `createScenes()` function to create a list of all of the `Scene` objects to be assigned to locations in the map. Finally, we call the `assignScenesToLocations()` function to assign the scenes to various locations in the map. The map is then returned to the `createNewGame()` function.

```
private static Map createMap() {
    // create the map
    Map map = new Map(20, 20);

    // create the scenes for the game
    Scene[] scenes = createScenes();

    // assign scenes to locations
    GameController.assignScenesToLocations(map, scenes);

    return map;
}
```

Next, we created stubs for these three functions and then implement each of the functions.

Instructions:

Implement the `createMap()` function for your game.

Overload the constructor Map () function

First, we will create the overloaded Map(int rows, int columns) function in the Map class. Here is the algorithm for the overloaded map function.

```
Map(rows, columns)
BEGIN
    IF rows < 1 OR columns < 1 THEN
        DISPLAY "rows and columns must be > 0" RETURN
    ENDIF

    Save the number of rows and columns
    Create a 2-D array of Locations

    FOR every row in the array
        FOR every column in the array
            Create and initialize a new Location object
            locations[row][column] = location
        ENDFOR
    END
END
```

We first check to see if an invalid number of rows and columns being was passed to the function. The we save the number of rows and columns in the map and then create a two dimensional array to contain all of the locations in the map. Finally, we iterate over every position in the array. A location object is created and assigned to each position in the array.

Here is the code for the overloaded map constructor function.

```
public Map() {  
}  
  
public Map(int noOfRows, int noOfColumns) {  
    if (noOfRows < 1 || noOfColumns < 1) {  
        System.out.println("The number of rows and columns must be > zero");  
        return;  
    }  
  
    this.noOfRows = noOfRows;  
    this.noOfColumns = noOfColumns;  
  
    // create 2-D array for Location objects  
    this.locations = new Location[noOfRows][noOfColumns];  
  
    for (int row = 0; row < noOfRows; row++) {  
        for (int column = 0; column < noOfColumns; column++) {  
            // create and initialize new Location object instance  
            Location location = new Location();  
            location.setColumn(column);  
            location.setRow(row);  
            location.setVisited(false);  
  
            // assign the Location object to the current position in array  
            locations[row][column] = location;  
        }  
    }  
}
```

Instructions:

Create an overloaded Map() constructor function for your program to create and initialize a two dimensional array of Location objects for your game.

Implement the `createScenes()` function

Next, we implemented the `createScenes()` function. Here is the algorithm for the `createSceneList()` function. This function is very similar to the `createActorList()` function described earlier except that you are creating a list of `Scene` objects instead of a list of `Actor` objects.

```
createSceneList()
BEGIN
    Create the scenes array

    Create scene 1
    Assign scene 1 to array
    ...
    Create scene N
    Assign scene N to array

    return scenes array
END
```

The implementation of the `createScenes()` function is very similar to the `createActorList()` function earlier. We first start by getting a reference to the `Game` object saved in the `CuriousWorkmanship` class. Then we create an array of `Scene` objects and then created each individual `Scene` object and assigned it to the next position in the `scenes` array. The creation of all the scenes is not shown for brevity. Also, the creation of the image objects for each scene is beyond the scope for this lesson. Do not worry about implementing it in your program.

```

private static Scene[] createScenes() {
    Game game = CuriousWorkmanship.getCurrentGame();

    Scene[] scenes = new Scene[SceneType.values().length];

    Scene startingScene = new Scene();
    startingScene.setDescription(
        "\nAnd we did come to the land which was called Bountiful, "
        + "because of its much fruit and also wild honey; and all "
        + "these things were prepared of the Lord that we might not "
        + "perish. And we beheld the sea, which we called Irreantum, "
        + "which, being interpreted, is many waters");
    startingScene.setMapSymbol(" ST ");
    startingScene.setBlocked(false);
    startingScene.setTravelTime(240);
    scenes[SceneType.start.ordinal()] = startingScene;

    Scene finishScene = new Scene();
    finishScene.setDescription(
        "\nCongratulations! Well done thou good and faithful servant."
        + "You have just launced your ship of curious workmanship and "
        + "\nbegun your journey to the promised land.");
    finishScene.setMapSymbol(" FN ");
    finishScene.setBlocked(false);
    finishScene.setTravelTime(Double.POSITIVE_INFINITY);
    scenes[SceneType.finish.ordinal()] = finishScene;
}

```

```

public enum SceneType {
    start,
    angel,
    lumber,
    iron,
    rye,
    rice,
    wheat,
    oliveOrchard,
    lentil,
    fababeans,
    water,
    honey,
    salt,
    cliff,
    ocean,
    desert,
    river,
    finish
}

```

Notice also that we created and used the `SceneType` enum list to represent the index positions of each `Scene` object in the `Scenes` array. Here is the partial implementation of the `SceneType` enum list. Again, some of the scenes have been omitted for brevity.

Instructions:

Create the `createScenes()` function for your program.

Implement the `assignScenesLocation()` function

The `assignScenesToLocations()` function is responsible for assigning each of the scenes just created to the various locations in the map. A portion of the function is shown below.

```
private static void assignScenesToLocations(Map map, Scene[] scenes) {
    Location[][] locations = map.getLocations();

    // start point
    locations[0][0].setScene(scenes[SceneType.desert.ordinal()]);
    locations[0][1].setScene(scenes[SceneType.desert.ordinal()]);
    locations[0][2].setScene(scenes[SceneType.start.ordinal()]);
    locations[0][3].setScene(scenes[SceneType.beach.ordinal()]);

    locations[9][9].setScene(scenes[SceneType.ocean.ordinal()]);
}
```

We first get the two dimensional locations array stored in the map. We then call the `setScene()` setter function and pass it a particular scene in the `scenes` array passed in. Notice that we specified the index of the position of the scene in the scene array using `SceneType` enum list item that corresponds to the scene being assigned to the location.

Instructions:

Create the `assignScenesToLocations()` function for your program.

Implementing the Game Menu options

The Game Menu is displayed after the a new game is created in our program. We need to start implementing the menu options in Game Menu next. Here is the Game Menu for the example program.

Game menu

The computer then displays the map of the land of Bountiful and the game menu below if a valid value is entered.

```
V - View map
I - View list of items in inventory
A - View list of actors
S - View ship status
L - View contents of location|
M - Move person to new location
E - Estimate the resource needed
B - Design barrels
C - Construct tools
R - Harvest resource
D - Deliver resource
W - Work on ship
P - Pack ship
J - Launch ship
H - Help
Q - Quit
```

The user enters a choice and the computer switches to the selected view.

We will begin by implementing the View list of items in inventory option. The GameMenuView class in the View Layer package is responsible for handling all input request by the end user. We will implement this option in the GameMenuView class.

We created and called stub functions to handle each of the options selected by the end user in the doAction() function of the GameMenuView class as shown below. The viewInventory() function is responsible for building the output view to display the list of inventory items.

```
public void doAction(char selection) {  
    switch (selection) {  
        case 'V': // Travel to new location  
            this.displayMap();  
            break;  
        case 'I': // View list of items in inventory  
            this.viewInventory();  
            break;  
        case 'A': // View list of actors  
            this.viewActors();  
            break;  
        case 'S': // View the ship's status  
            this.viewShipStatus();  
            break;  
        // ...  
    }  
}
```

```
private void viewInventory() {  
    System.out.println("*** viewInventory stub function called ***");  
}
```

We implement this function by first creating the algorithm for the `viewInventory()` function.

```
viewInventory(): void
BEGIN
    Get the sorted list of inventory items
    DISPLAY "List of Inventory Items"
    Display the column headings
    FOR every item in the inventory list
        DISPLAY i.description, item.quantityInStock
    ENDFOR
END
```

Then we copied the algorithm into the `viewInventory()` function and translated each of the statements into Java code. Here is the code for the `viewInventory()` function.

```
private void viewInventory() {
    // get the sorted list of inventory items for the current game
    InventoryItem[] inventory = GameControl.getSortedInventoryList();

    System.out.println("\nList of Inventory Items");
    System.out.println("Description" + "\t" +
        "Required" + "\t" +
        "In Stock");

    // for each inventory item
    for (InventoryItem inventoryItem : inventory) {
        // DISPLAY the description, the required amount and amount in stock
        System.out.println(inventoryItem.getDescription() + "\t" +
            inventoryItem.getRequiredAmount() + "\t" +
            inventoryItem.getQuantityInStock());
    }
}
```

First, we call a Control Layer function to get a sorted list of inventory items. We decided to put this function in the `GameControl` class because it is getting and returning data objects which is one of the responsibilities of the Control Layer. We created a stub for this function for now. It will be implemented later.

```
public static InventoryItem[] getSortedInventoryList() {
    System.out.println("\n*** getSortedInventoryList stub function called ***");
    return null;
}
```


The next two statements in the `viewInventory()` function print the title and column headings for the report. Finally, we used a `for-each` statement to iterate through each of the items in the inventory list and print out the description, required amount and number of items in stock.

Implement the `getSortedInventoryList()` function

We need to implement the `getSortedInventoryList()` function in the `GameControl` class next. Here is the algorithm for the `getSortedInventoryList()` function.

```
getSortedInventoryList(inventoryList): sortedArray
BEGIN
  get the inventory list from the current game
  clone inventory list
  FOR (i=0; i < listLength; i++)
    FOR (j=0; j < listLength-1-i; j++)
      IF item[j].description > item[j+1].description
        swap the positions of the two items
      ENDIF
    ENDFOR j
  ENDFOR i

  RETURN cloned sorted inventory list
END
```

This function will get and clone the list of inventory items created and saved in the game. “Clone” in programming refers to making a copy of the original object. We cloned the original list so that we do not want to sort the sort and change the order of the items in the original list. Instead we will return a copy of the original list in sorted order. A bubble sort is then to alphabetize the inventory items by their description.

Now we need to translate the algorithm into Java code. The first statement in the algorithm requires that we get the original inventory list stored in the current game earlier. We created the inventory list and saved it in the current game in the `createNewGame()` function implemented earlier as shown below.

We must first get the current game first before we can get the inventory list that we saved in the game. The current game was saved in the the class (`CuriousWorkmanship`) containing the `main()` function. We reference the `CuriousWorkmanship` class and call it's static `getCurrentGame()` function to get the current game and then call the game's `getInventory()` function to get the inventory list saved in the game.

```
public static InventoryItem[] getSortedInventoryList() {  
    // get inventory list for the current game  
    InventoryItem[] originalInventoryList =  
        CuriousWorkmanship.getCurrentGame().getInventory();  
}
```

The next statement in the algorithm is to clone the original list. Every object in Java has a `clone()` function. We called the `clone()` function to create a copy of the `originalInventoryList`.

```
public static InventoryItem[] getSortedInventoryList() {  
    // get inventory list for the current game  
    InventoryItem[] originalInventoryList =  
        CuriousWorkmanship.getCurrentGame().getInventory();  
  
    // clone (make a copy) originalList  
    InventoryItem[] inventoryList = originalInventoryList.clone();  
}
```

Next we need to write the code to implement the bubble sort algorithm to sort the list of inventory items. Here is the full implementation of the `getSortedInventoryList()` function after implementing the bubble sort.

```
public static InventoryItem[] getSortedInventoryList() {  
    // get inventory list for the current game  
    InventoryItem[] originalInventoryList =  
        CuriousWorkmanship.getCurrentGame().getInventory();  
  
    // clone (make a copy) originalList  
    InventoryItem[] inventoryList = originalInventoryList.clone();  
  
    // using a BubbleSort to sort the list of inventoryList by name  
    InventoryItem tempInventoryItem;  
    for (int i = 0; i < inventoryList.length-1; i++) {  
        for (int j = 0; j < inventoryList.length-1-i; j++) {  
            if (inventoryList[j].getDescription().  
                compareToIgnoreCase(inventoryList[j + 1].getDescription()) > 0) {  
                tempInventoryItem = inventoryList[j];  
                inventoryList[j] = inventoryList[j+1];  
                inventoryList[j+1] = tempInventoryItem;  
            }  
        }  
    }  
  
    return inventoryList;  
}
```

We used nested `for` statements to implement the sort. The outer `for` statement goes through every position in the list. For each item in the list, the inner `for` loop goes through the entire list and compares the description of the current inventory item (`j`) to see if it is greater than the description of the inventory item at the next position (`j+1`) in the list. The position of the two inventory items are then swapped if the current if the result of the comparison is positive (i.e., greater than zero).

The `compareToIgnoreCase()` `String` function is used to compare the two strings. It ignores any differences in the use of upper and lower case letters in the two strings. Here is the function signature for the function:

```
public int compareToIgnoreCase(String str)
```

It returns a positive value if the this string is alphabetically greater than the string passed to the `str` input parameter, a zero if the two strings are equal and a negative value if the first string is less than the second string.

Here is a simple example of using the `compareToIgnoreCase()` function. The value of `String` variable `fred` is compared to the value of variable `bob`. The value of `fred` is alphabetically greater than the value of `bob` so the value returned and assigned to the `result` variable will be positive.

```
String fred = "Fred";  
String bob = "Bob"  
int result = fred.compareToIgnoreCase(bob);
```

Instructions

Create a Control Layer function to sort one of the list of objects in your program. Use a different sort algorithm other than the bubble sort shown above. Here is a link to view other types of commonly sorting algorithms that you might choose to implement for your sort function.

[Commonly used sorting algorithms](#)

Then implement a function in the View Layer to display the sorted list of objects when the player selects an option on one of the menus in your program. Run and test your program to make sure the list will display properly.

Implement the view map option in the Game Menu

The first option in the Game Menu is to view or display the map. We need to modify the `GameMenuView` class to implement this option.

A stub function called `displayMap` is created and called in the `doAction()` function of the `GameMenuView` class to handle the end users request to view the map.

```
public void doAction(char selection) {  
    switch (selection) {  
        case 'V': // Travel to new location  
            this.displayMap();  
            break;  
        case 'I': // View list of items in inventory  
            this.viewInventory();  
            break;  
        case 'A': // View list of actors  
            this.viewActors();  
            break;  
        case 'S': // View the ship's status  
            this.viewShipStatus();  
            break;  
        // ...  
    }  
}
```

```
public void displayMap() {  
    System.out.println("*** displayMap stub function called ***");  
}
```

We need to now implement the `displayMap()` function to create the output view to display the contents for every location of the map.

Here is the algorithm for the `displayMap()` function.

```
displayMap(locations): void
BEGIN
    get the map locations from the current game
    DISPLAY title
    DISPLAY row of column numbers
    FOR every row in map
        DISPLAY row divider
        DISPLAY row number
        FOR every column in row
            DISPLAY column divider
            location = locations[row][column]
            IF location has been visited
                DISPLAY the map symbol for location
            ELSE
                DISPLAY " ?? "
            ENDIF
            DISPLAY ending column divider
        ENDFOR
        DISPLAY ending row divider
    END
END
```

We first get the two-dimensional array of `Location` objections from the map. We then display the title. Next, we display the column headers for each column in the map. We then iterate through every row of the map. For each row we print a row divider (e.g., a row of dashes). We then move down to the next line and print the row number. Then we use a for repetition to go through every column in the map. For each column, we print a column divider (" | "). We get the `Location` object at the current row and column. If the `Location` has been visited then we display a symbol that indicates what the contents of location is (i.e., "~~~~" represents an ocean scene. If the location has not been visited then we print " ?? " to indicate that this location has not been visited yet. After the last column has been displayed, we print a final column divider. We then repeat and get the next row of the map. The process continues until all of the rows have been printed. Finally, we print another row divider at the bottom of the map.

Instructions:

Implement a View Layer function to display the map in your program. Use the algorithm for the `displayMap()` function above to create as the basis for your function. Run and test your program to make sure that it displays the map correctly.

Submit your assignment

1. Be sure to save all of your changes and commit your changes to the local repository. Pull from the remote GitHub repository and merge any changes into your local repository. If conflicts occur, resolve the conflicts and save your changes. Commit the changes to your local repository again. Finally, Push your code to the remote GitHub repository.
2. Submit your assignment and add a note with the name of your team member/s, the url of your repository and a list of the classes that your team modified.