GameWorld API Specification

Introduction

This document serves as the starting guide to implementing the GameWorld API. The API links two "types" of applications; the game world and the controlling application. The game world is a set of stand-alone classes that cannot be run on their own. They need to be passed on to a controlling application (like Blockr) in order to function. The controlling application provides a `java.awt.Graphics` object to the game world so it can render on the canvas window, and it tells the game world what actions to execute.

This guide will use IntelliJ IDEA as the IDE in our examples, but the API can also be used in other IDEs, like Eclipse.

Getting started

Building the API

To be able to use the API, we need to build a JAR file that contains all the API interface classes. To do that, open the `gameworldapi` project in IntelliJ and go to File → Project Structure → Artifacts → + → JAR and select *From modules with dependencies*. Select `gameworldapi.GameWorldType` as your Main Class.

Next, go to Build → Build Artifacts... → gameworldapi:jar → Build and build the JAR file. It will be located in

gameworldapi/out/artifacts/gameworldapi_jar/gameworldapi.jar

Including the API in the game world project

To start implementing the API in your game world project, open the project and go to File → Project Structure → Libraries → + → Java and select the *gameworldapi.jar* file. You will now need to implement the following interfaces in the project to fulfill the API requirements:

1. GameWorldType
2. ActionType
3. PredicateType
4. GameWorld
5. GameWorldState

If you also need some way of specifying coordinates, you can implement the `Location` interface as an *immutable* class. More details on how what the implementation requirements are are found in the relevant section below.

Including the API in the controlling application

The controlling application (e.g. Blockr) should also implement the API in the same way as the game world project: go to File → Project Structure →

Libraries → + → Java and select the *gameworldapi.jar* file. The controlling application should also be able to communicate with a game world. This should be done by passing the game world's GameWorldType class file as an argument to the controlling application. Here is some example code to demonstrate an implementation of the class loading:

```
public static void main(String[] args) {
   try{
       File file = new File(args[0]);

       //convert the file to URL format
       URL url = file.toURI().toURL();
       URL[] urls = new URL[]{url};

       //load this folder into Class loader
       ClassLoader cl = new URLClassLoader(urls);

       //load the class passed as a second argument
       Class cls = cl.loadClass(args[1]);
       GameWorldType worldType = (GameWorldType) cls.newInstance();
       java.awt.EventQueue.invokeLater(() -> {
           new MyCanvasWindow("Window title", worldType).show();
       }); }
   catch (Exception ex){
       System.out.println("Error: " + ex.getMessage().toString());
   }
}
```

The class can then be loaded by passing two arguments:

1. The path of the class file, excluding the package
2. The class file, in the format of `package.ClassName`

For example:

> /home/jesse/project/gameworld/out/production/gameworld gameworld.GameWorldType

Implementation requirements

Game world

GameWorldType The GameWorldType interface acts as a Facade to the controlling application. Its implementation is the class that will be passed on to the controlling application as an argument. The GameWorldType class that implements the GameWorldType interface must have three methods:

1. `public ArrayList<ActionType> getSupportedActions();` This function should return an ArrayList containing all the supported Actions. An

Action is an implementation of an ActionType, and is specified below. Some example code: `return new ArrayList<>(Arrays.asList(Action.TURN_LEFT, Action.GO_UP));`

2. `public ArrayList<PredicateType> getSupportedPredicates();` This function works the same as above, except that it now returns the supported Predicates instead of the supported Actions. Just as with the Actions, a Predicate is an implementation of a PredicateType and is specified below.

3. `public GameWorld newWorldInstance();` This function should return a new instance of the implementation of the GameWorld interface. In the controlling application, this method should only be called once. Otherwise, multiple instances of games would be created which is not wanted behaviour.

Actions & Predicates Your game world implementation should have an `Action` enum and a `Predicate` enum that implement the `ActionType` and `PredicateType` interface respectively. An Action represents something (an action) that can be performed on the game world and results in changing the state of the game world. A predicate is a statement about the state of the game world that can either be true or false. An example of an action is moving the player forward, an example of a predicate is "player is in front of wall".

Actions and predicates have to be implemented as en enum, with the literals representing the possible actions and predicates.

A code example:

```
import gameworldapi.*;
public enum Action implements ActionType {
    MOVE_FORWARD, TURN_LEFT, TURN_RIGHT;
}
```

GameWorld The GameWorld class that implements the GameWorld interface is arguably the most important class of your project. It represents the whole game world and is used to interface with the game. The GameWorld class must have at least the following methods in its body:

1. `ActionResult perform(ActionType action);` This function takes an ActionType as an argument and performs the action on the current gameworldstate. It returns an ActionResult indicating whether the action was successful or not and whether the game has ended now or not.

   If performing the action was successful, the gameworldstate should be updated accordingly.

2. `Boolean evaluate(PredicateType predicate);` This function evaluates a predicate in the GameWorld's current state and subsequently returns a boolean indicating the evaluation of the predicate.

3. `GameWorldState getSnapshot();` This function returns an immutable snapshot of the GameWorld's current state.
4. `void restore(GameWorldState gameWorldState);` This function takes a snapshot of the GameWorld's state as an argument and updates its state to this snapshot. It is equivalent to `setState(GameWorldState state);`
5. `void render(Graphics g, int x, int y);` This function triggers the rendering of the Game World on the controlling applications canvas window. It needs a `java.awt.Graphics` object to paint on. `int x` and `int y` represent the x and y-coordinates of the top-left corner (in px) of the GameWorld on the CanvasWindow where the drawing should begin.

ActionResult The ActionResult is an enum that represents the four possible outcomes of performing an Action in the GameWorld:

1. **SUCCESS** this means that the action was performed successfully and that the GameWorldState has been updated to reflect the action. The game is thus still going on.
2. **FAILURE** this means that the action could not be performed for some reason. The GameWorldState has not been updated, but the game is still running.
3. **GAME_OVER** this means that the action that was executed has lead to the end of a game in a negative way, e.g. the player has died. In the controlling application, gameworld should be reset at the next execution.
4. **GAME_SUCCESS** this means that the gamehas ended successfully; the player won the game. The gameworldstate should be reset after this.

GameWorldState The GameWorldState class that implements the GameWorldState interface represents the state of the game world at a certain point in time. It must be an **immutable** class. It should contain everything that can change during gameplay, like a player's location or direction. It is advised that this class is created by a special Factory class.

Location The Location interface is an optional interface that can be implemented by a Location class if you need a way to use two-dimensional coordinates in your program. It is very important that, if you decide to use this interface, you implement it as an immutable class.

Controlling application

In the section above, you have seen how to include the API in your project and how to load the game world dynamically. In order for all this to function properly, there are some requirements that must be fulfilled.

On startup, the controlling application will receive an instance of a GameWorld-Type class. The controlling applications should poll the game world for its supported actions and predicates, and should then offer the user the necessary controls for executing these actions and evaluating the predicates. This can be done with the `ActionResult perform(ActionType action)` and `Boolean evaluate(PredicateType predicate)` functions in the game world.

It should also ask for a new GameWorld instance **once**, on startup. It should store the GameWorld object in a logical way so that it can be used throughout the program.

It is advised that, on startup, the controlling application asks the game world for its current state as this is its initial state. This allows the controlling application to reset the game when it deems this necessary, by calling the `void restore(GameWorldState state)` function in the GameWorld class, and then passing the initial GameWorldState as its argument.

Furthermore, when the user interacts with the program and triggers a UI update, the controlling application should call the `void render(Graphics g, int x, int y)` function in the game world to trigger repainting the game world.

Just as the game world, you can also choose to implement the Location interface in your controlling application. Once again, this must be done as an immutable class.