CS4431 Text Adventure

Creating forgettable experiences with high quality code

Roman Moisieiev – 25442015

Most code shortened/omitted for brevity.

Content Warning: Concurrency, JavaFX, Premature Abstraction.

Custom Data Structures

2D Array

```java
public class Matrix<T> implements Iterable<MatrixElement<T>> {
    private final ArrayList<T> storage;
    final int width;
    final int height;

    public Matrix(int width, int height);
    public void set(int row, int column, T value);
    public T get(int row, int column);
    public List<T> row(int row);
    public Iterator<MatrixElement<T>> iterator();
}
```

Implemented as a dense-array, providing the lowest memory overhead and great cache locality, at the cost of slow resizing.

2D Array iterator with positional context

```
class MatrixIterator<T> implements Iterator<MatrixElement<T>>;

public class MatrixElement<T> {
    private final int row, column;
    public T value;

    public int getRow();
    public int getColumn();
}
```

This allowed me to easily write position-aware code on top of the Matrix, which I used for building the map layout using DFS.

Blocking Double-Ended Queue

```java
public class BlockingArrayListDeque<T> {
    private final ArrayList<T> list;
    private int len = 0;
    private int start = 0;

    public BlockingArrayListDeque(int capacity);

    public int size();
    public void push_back(T item) throws InterruptedException;
    public void push_front(T item) throws InterruptedException;
    public T pop_front() throws InterruptedException;
    public T pop_back() throws InterruptedException;
}
```

Implemented using a ring-buffer on top of `ArrayList`, filling empty slots with null. All operations are O(1) amortized runtime.

Completion Trie

```
public class CompletionTrie {
    TrieNode root = new TrieNode("");

    void insert(String word);
    void insertAll(String... words);
    void delete(String word);

    ArrayList<String> search(String word);
}
class TrieNode {
    TreeMap<Character, TrieNode> children;
    boolean isEnd = false;
    String prefix;
}
```

Allows for O(n) completion that does not depend on the number of registered commands.

Command Handling Design

Commands can freely modify the state of the game instance.

A Command is produced by invoking `CommandParser::parse`.

```
public abstract class Command {
    abstract void execute(ZorkInstance instance) throws CommandException;
}
```

A `CommandParser` exhaustively describes any given command’s external interface, erasing the need for any special-case behavior for any command.

```
interface CommandParser {
    // Empty signifies a parsing failure
    Optional<Command> parse(String text);

    void autoComplete(GameState context, ArrayList<String> output, String text);
    // Used to register top-level commands into the CompletionTrie
    void registerDirectCompletions(CompletionTrie trie);

    String getName();
    String getDescription();
}
```

CommandParsers are managed by the `CommandRegistry` singleton.

```java
public class CommandRegistry {
    public final static CommandRegistry INSTANCE = new CommandRegistry();

    private final ArrayList<CommandParser> commandParsers = new ArrayList<>();
    private final CompletionTrie completionTrie = new CompletionTrie();

    public static String describeCommands();
    private static void registerParser(CommandParser parser);
    public static List<String> autocomplete(GameState context, String text);
    public static Optional<Command> parse(String text);
}
```

```
public class Item {
    @JsonProperty("description")
    private String description;
    @JsonProperty("name")
    private String name;
    @JsonIgnore
    private String id;

    public void useInInventory(GameState context);
    public void useInRoom(GameState context);
    public void pickUp(GameState context);
    public void drop(GameState context);
    public String getDescription();
    public String getName();
}
```

The Item class is extended by every item, and stored in a special `TypedItems` container, this way its real subclass can be retained and safely accessed without casts.

```java
public class TypedItems {
    @JsonProperty("keys")
    final Keys keys = new Keys();

    @JsonProperty("computer")
    final Computer computer = new Computer();

    public Map<String, Item> toItemMap();
}
```

At runtime, once a save file is loaded, the typed items are loaded into a `Map<_, Item>`, allowing every API not aware of the Item’s special properties to address it as a simple Item, as a manual form of type-erasure.

Same approach is used for rooms with special behavior.

```
public class SaveManager {
    public static Path getSaveDirectory();
    public static Optional<File[]> listSaveFiles();
    public static List<String> listSaveNames();
    public static Path pathForSaveName(String name);
    public static GameState loadState(String name) throws JacksonException;
    public static Optional<GameState> loadInitialState(String save_name);
    public static void saveState(String name, GameState game) throws JacksonException;
}
```

Save files are stored in XDG directory standard compliant locations, as JSON files.

UI: An interface in more ways than one.

```java
public interface ViewController {
    boolean WasExitRequested();
    void notifyOfCompletion();
    <T> Optional<T> presentSelectionList(List<T> options);
    String presentTextSelectionListWithPrompt(List<String> options, String prompt);
    Optional<String> consumeTextInput();
    void presentTextPrompt(String prompt);
    void presentMessage(String message);
    void presentUrgentMessage(String message);
    void presentErrorMessage(String message);
}
```

The entire program is generic over the interface used to communicate with the user, which is used to select between a JavaFX GUI and a Terminal interface using a command-line flag(`--cli`).