The University of Queenland, School of ITEE
CSSE2002/7023 — 2018
Assignment 2 (15%)
Due: October 2, 2018
Revision: 1.3 (as at September 16, 2018)

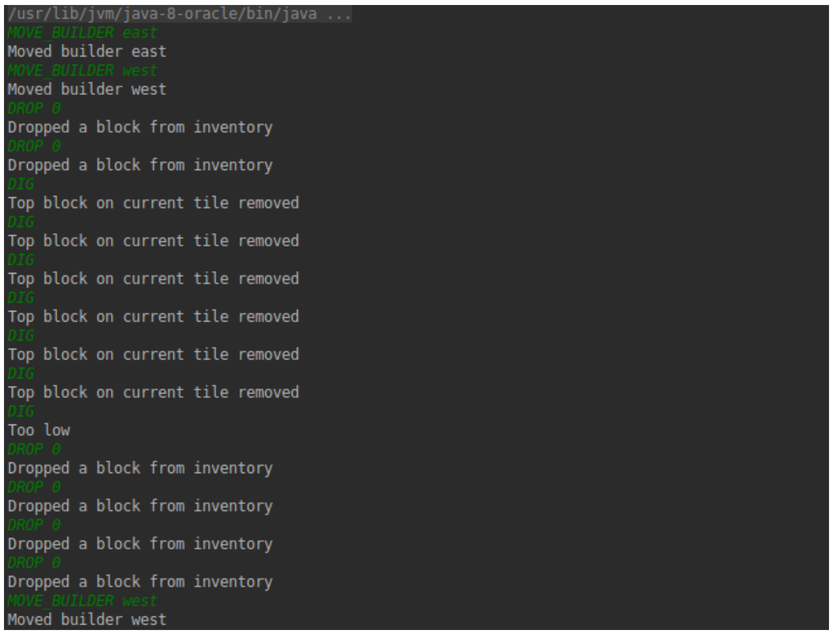
Introduction

The goal of this assignment is to add a set of classes and methods to those developed in Assignment 1. You will implement precisely the public and protected items described in the supplied documentation (no extra members or classes). Private members will be for you to decide.
Language requirements: Java version 1.8, JUnit 4

Context

Now that we have developed a set of primitives to represent Tile, Block, and Builder, we want to be able to:

- arrange tiles into a world map,
- load world maps,
- load or enter actions that a builder can perform,
- run the actions, and
- save the updated world map.



```
/usr/lib/jvm/java-8-oracle/bin/java ...
MOVE_BUILDER east
Moved builder east
MOVE_BUILDER west
Moved builder west
DROP 0
Dropped a block from inventory
DROP 0
Dropped a block from inventory
DIG
Top block on current tile removed
DIG
Top block on current tile removed
DIG
Top block on current tile removed
DIG
Top block on current tile removed
DIG
Top block on current tile removed
DIG
Top block on current tile removed
DIG
Too low
DROP 0
Dropped a block from inventory
DROP 0
Dropped a block from inventory
DROP 0
Dropped a block from inventory
DROP 0
Dropped a block from inventory
MOVE_BUILDER west
Moved builder west
```

Figure: A simple command interface for entering actions that the builder can perform.

In this assignment you will create classes to represent the world map, actions, and a main function for your application.

Ethical obligations

All work on this assignment is to be your own individual work. As detailed in Lecture 1, code supplied by course staff is acceptable but there are no other exceptions.

You are expected to be familiar with "What not to do" from Lecture 1 and `http://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism`.

If material is found to be "lacking academic merit", that material may be removed from your submission prior to marking. No attempt will be made to repair any code breakage caused by doing this.

If you have questions about what is acceptable, please ask.

Supplied material

- This task sheet
- A `.zip` file containing html documentation (Javadoc) for the classes and interfaces you are to write (also on Blackboard). Unzip the bundle somewhere and start with `doc/index.html`.
- A `.zip` file containing implementations of the Assignment 1 classes (we have moved these classes into a new package).

Tasks

1. Implement each of the following classes and interfaces (described in the Javadoc):
   - `csse2002.block.world.Action`
   - `csse2002.block.world.ActionFormatException`
   - `csse2002.block.world.Main`
   - `csse2002.block.world.Position`
   - `csse2002.block.world.SparseTileArray`
   - `csse2002.block.world.WorldMap`
   - `csse2002.block.world.WorldMapFormatException`
   - `csse2002.block.world.WorldMapInconsistentException`
2. Write JUnit4 tests for the methods in the following classes:
   - `csse2002.block.world.Action` as `csse2002.block.world.ActionTest`
   - `csse2002.block.world.SparseTileArray` as `csse2002.block.world.SparseTileArrayTest`

Marking

The 100 marks available for the assignment will be divided as follows:

Symbol	*Marks*	*Marked*	*Description*
F	55	Electronically	Implementation and functionality: Does the submission conform to the documentation?
S	25	By "humans"	Style and clarity.
J	20	Electronically	Student supplied JUnit tests: Do the tests correctly distinguish between correct and incorrect implementations?

The overall assignment mark will be $A_1 = F + S + J$ with the following adjustments:

1. If $F < 5$, then $S = 0$ and $J = 0$ and "style" will not be marked.
2. If $S > F$, then $S = F$.
3. If $J > F$, then $J = F$.

For example: $F = 22, S = 25, J = 17 \Rightarrow A_1 = 22 + 22 + 17$.

The reasoning here is not to give marks to cleanly laid out classes which do not follow the specification.

Functionality marking

The number of functionality marks given will be

$$F = \frac{\text{Tests passed}}{\text{Total number of tests}} \cdot 55$$

Each of your classes will be tested independently of the rest of your submission. Other required classes for the tests will be copied from a working version. Functionality testing does not apply to your JUnit tests.

Note: Where "cannot be X" (e.g., "cannot be null") is used in the specification, it indicates that X is not a valid input to the function and the function will not be tested with input X.

Style marking

As a style guide, we are adopting[1] the Google Java Style Guide `https://google.github.io/styleguide/javaguide.html` with some modifications:

- 4.2 Indenting is to be +4 chars not +2.
- 4.4 Column limit for us will be 80 columns.
- 4.5.2 First continuation is to be +8 chars.
  - All private/"package private" members must be commented (you may use Javadoc comments but are not required to).
  - Java source files must be encoded as either ASCII or UTF-8.
  - All public and protected comments are expected to use Javadoc markup (e.g. @param, etc).

[1]There is no guarantee that code from lectures complies.

There is quite a lot in the guide and not all of it applies to this course (eg no copyright notices). The marks are broadly divided as follows:

Naming	5
Commenting	6
Structure and layout	8
Good OO implementation practices	6

Note that this category does involve some aesthetic judgement (and the marker's aesthetic judgement is final).

Commenting

All functions and member variables need to be commented for this assignment. Comments for functions need to explain how the function is used, what the function returns, and what changes the function makes to its calling instance.

All public and protected member functions and variables must have Javadoc comments.. For functions and variables specified in the assignment, descriptions can be taken from the specification and added as comments.

Additionally, any section of code (inside a function) that would be difficult for another reader to understand needs comments that explain what the code does.

Test marking

Marks will be awarded for test sets which distinguish between correct and incorrect implementations[2]. A test class which passes everything (or fails everything) will receive a mark of zero.

There will be some limitations on your tests:

1. If your tests take more than 20 seconds to run, they will be stopped and a mark of zero given.

2. Each of your test classes must be less than 800 (non-empty) lines. If not, that test will not be used.

3. ***Your tests must only call functions specified in the Javadocs for classes that you are testing.***

These limits are very generous, (eg your tests shouldn't take anywhere near 20 seconds to run).

Electronic Marking

The electronic aspects of the marking will be carried out in a virtual machine. The VM will not be running Windows and neither IntelliJ nor Eclipse will be involved. For this reason, it is important that you name your files correctly. ***It is also important that you do not import any non-standard packages.***

It is critical that your code compiles. If one of your classes does not compile, you will recieve zero for any electronically derived marks for that class.

It is critical that your class, interface and public member names match the Javadocs. If the names do not match, you could lose marks for all functionality associated with that class, interface or member. No corrections of typos will be made by staff. It is your responsibility to ensure that your submission is error-free and meets with the specification.

[2] And get them the right way around

Submission

Submission is via the course blackboard area `Assessment/Ass2/Ass2 Submission`.

Your submission is to consist of a single `.zip` file with the following internal structure:

`src/csse2002/block/world`	`.java` files for classes described in the Javadoc
`test/csse2002/block/world`	`.java` files for the test classes

A complete submission would look like:

```
src/csse2002/block/world/Action.java
src/csse2002/block/world/ActionFormatException.java
src/csse2002/block/world/Main.java
src/csse2002/block/world/Position.java
src/csse2002/block/world/SparseTileArray.java
src/csse2002/block/world/WorldMap.java
src/csse2002/block/world/WorldMapFormatException.java
src/csse2002/block/world/WorldMapInconsistentException.java
test/csse2002/block/world/ActionTest.java
test/csse2002/block/world/SparseTileArrayTest.java
```

Any submission which does not comply with this structure will receive a penalty of 10% of the maximum marks available. Your classes must declare themselves to be members of the `csse2002.block.world` package (both sources and test sources). Do not submit any other files (eg no `.class` files). Remember that java filenames are case sensitive when your filesystem isn't.

Late submission

As stated in the ECP:

- No late submissions will be accepted.
- Requests for extensions must be made at least 48 hours prior to the submission deadline. Extension requests received after this point may not be able to be considered.
- Due to the incremental nature of the assessment items in this course, extensions to Assignment 2 are extremely unlikely. If you apply for an extension, you may be asked to meet with the course coordinator to discuss alternatives. Failure to attend this meeting before the assignment due date will result in your extension request being denied, unless the medical or other circumstances are such that you could not reasonably be expected to attend the meeting.

Revisions

If it becomes necessary to correct or clarify the task sheet or Javadoc, a new version will be issued and a course announcement will be made on blackboard. No changes will be made on or after Monday of Week 12 (i.e. September 17, 2018). ***This means that you need to periodically check blackboard until this date.***

Version 1.1 — September 12, 2018

- Changed `ActionFormationException.java` to `ActionFormatException.java`
- Changed assignment weight to be the same as the ECP
- Made the following changes to Javadoc spec:
  - Changed all exit names "north", "east", "south" and "west" to be lower case throughout spec.
  - "Error: Invalid Action" to "Error: Invalid action" (Note lower case 'a')
  - Added instructions for handling empty inventory, and tiles with no blocks or no exits for the block world map format.
  - Added to the specification of addLinkedTiles about geometric inconsistency.

Version 1.2 — September 13, 2018

- Changed due date to include extension
- Made the following changes to Javadoc spec:
  - Change spec for `Action.loadAction()` to throw an `ActionFormatException` if DIG is not on a line by itself.
  - Change spec for `Action.loadAction()` so that it should not check for valid secondary actions.
  - Reformatted spec for `Action.loadAction()`.
  - Changed spec for `Action.processAction()` to clarify that "Error: Invalid action" should be printed for a secondaryAction that is not an integer.
  - Changed spec for `Action.processAction()` to clarify that all valid integers given for DROP should be passed to `Builder.dropFromInventory()`.
  - Changed spec for `WorldMap.WoldMap(Tile, Position, Builder)` to require that the builder's current tile (`Builder.getCurrentTile()`) must be equal to the startingTile.
  - Changed spec for `WorldMap.WorldMap(String)` to require that a Builder is constructed such that the `Builder.getCurrentTile() == getTiles(0)`.
  - Changed spec for `WorldMap.WorldMap(String)` to require that a WorldMapFormatException is thrown if the Tile or Builder constructors throw an exception.
  - Changed spec for `WorldMap.WorldMap(String)` to clarify that the names of directions, blocks, and the id numbers should be checked as part of the formatting when loading the map.
  - Changed the spec for the constructor of SparseTileArray from "intialise an empty array" to "initialise an empty SparseTileArray".
  - Changed the spec for the constructor of SparseTileArray to require that SparseTileArray.getTiles() is an empty list.
  - Changed the spec for `SparseTileArray.getTiles()` to allow for the return of an immutable list or a list that cannot change the value of `SparseTileArray.getTiles()` in subsequent calls.

Version 1.3 — September 16, 2018

- Changed due date to include further extension
- Changed the line count limit for test cases to 800 lines (to encourage more clarity)
- Made the following changes to Javadoc spec:
  - Changed spec for `WorldMap.WorldMap(String)` to require that tile ids must be between 0 and the number of tiles - 1.
  - Changed spec for `WorldMap.WorldMap(String)` to require that the number of tiles must not be negative.
  - Changed spec for `WorldMap.WorldMap(String)` to require that if an IOException occurs, a WorldMapFormatException should be thrown.
  - Changed spec for `WorldMap.WorldMap(String)` to clarify behaviour of FileNotFoundException.
  - Changed spec for `Action.loadAction` to throw an ActionFormatException if the primary action is not one of the four specified in the format.
  - Fix typo in `WorldMap.WorldMap(String)`, starting tile lines should be 1 and 2.
  - Changed spec for `WorldMap.saveMap()` to change order of writing to be same as file format.
  - Corrected typo in `Action.loadAction()`, changed “2 spaces” to “2 or more spaces”.
  - Changed spec for `WorldMap.WorldMap(String)` to clarify that there should be no blank line at the end of a file (but may contain a single newline character).