Programming Fundamentals (48023) – Assignment

As only a portion of the class will do this assessment task, it would be wasteful to provide hardcopy to all students, and therefore this assignment will **NOT** be distributed in hardcopy at lectures (or anywhere else).

The following are extracts from the subject outline that are relevant to the assignment …

Assessment task 2: Assignment

Intent: The purpose of this assessment task is to provide students with the opportunity to show they can apply the basic skills and knowledge of programming in a context where it is not made explicit exactly which basic skills and knowledge need to be used.

Objective(s): This assessment task addresses the following subject learning objectives (SLOs):

1, 2, 3 and 4

This assessment task contributes to the development of the following course intended learning outcomes (CILOs):

B.1, B.2, B.5 and C.1

Groupwork: Individual

Weight: 30%

Task: In this assignment, students will build and/or extend a basic class structure. Students will submit their assignment for marking to the online PLATE system ("Peer Learning And Teaching Environment"). More details about the assignment will be provided in the assignment specification.

This assignment is an individual work.

Due: Assignment due at 11:59pm Sunday June 10, 2018 (i.e. 10/06/2018, 11:59pm). However, for feedback, students are encouraged to submit their partly completed assignments to PLATE regularly, prior to the deadline.

Assessment feedback

For the assignment, students receive feedback every time they submit their work to PLATE available at http://plate.it.uts.edu.au Students may also ask their tutor for help with the assignment during their weekly lab session.

Minimum requirements

Students must have completed all pass/fail tests and also all additional lab exercises (Assessment Tasks 1 and 3) for marks from the Assignment (Assessment Task 2) to be included in the aggregate mark.

Exemption to Minimum Requirements: To have their mark counted from Part A of the assignment, students do **NOT** have to complete the last TWO additional lab test on EACH thread (i.e. the four lab tests SelSortSizeN, ListOfNV3PartA, BubSortSizeN and ListOfNV3PartA). But students must complete those four lab tests to have marks counted for Parts B and C of the assignment. This exception allows students to commence working on Part A of the assignment before they have completed those last four lab tests, confident that their mark for Part A will count.

Extensions

When, due to extenuating circumstances, you are unable to submit or present an assessment task on time, please contact your subject coordinator before the assessment task is due to discuss an extension. Extensions may be granted up to a maximum of 5 days (120 hours). In all cases you should have extensions confirmed in writing.

Special Consideration

If you believe your performance in an assessment item or exam has been adversely affected by circumstances beyond your control, such as a serious illness, loss or bereavement, hardship, trauma, or exceptional employment demands, you may be eligible to apply for **Special Consideration**

https://www.uts.edu.au/current-students/managing-your-course/classes-and-assessment/special-circumstances/special

Late Penalty

Work submitted late without an approved extension is subject to a late penalty of 10 per cent of the total available marks deducted per calendar day that the assessment is overdue (e.g. if an assignment is out of 40 marks, and is submitted (up to) 24 hours after the deadline without an extension, the student will have four marks deducted from their awarded mark). Work submitted after five calendar days is not accepted and a mark of zero is awarded.

Plagiarism and academic integrity

At UTS, plagiarism is defined in Rule 16.2.1(4) as: 'taking and using someone else's ideas or manner of expressing them and passing them off as his or her own by failing to give appropriate acknowledgement of the source to seek to gain an advantage by unfair means'.

breaches of academic integrity that constitute cheating include but are not limited to:

- submitting work that is not a student's own, copying from another student, recycling another student's work, recycling previously submitted work, and working with another student in the same cohort in a manner that exceeds the boundaries of legitimate cooperation
- purchasing an assignment from a website and submitting it as original work
- requesting or paying someone else to write original work, such as an assignment, essay or computer program, and submitting it as original work.

Students who condone plagiarism and other breaches of academic integrity by allowing their work to be copied are also subject to student misconduct Rules.

Where proven, plagiarism and other breaches of misconduct are penalised in accordance with UTS Student Rules Section 16 – Student misconduct and appeals.

Work submitted electronically may be subject to similarity detection software. Student work must be submitted in a format able to be assessed by the software (e.g. doc, pdf (text files), rtf, html).

… *end of extracts from subject outline.*

Three Parts: A (worth 5%), B (worth 10%) and C (worth 15%)

There are **THREE** parts to the assignment, called "Part A" which is worth 5%, "Part B" which is worth 10%, and "Part C" which is worth 15%. All three parts have the same deadline. You do **NOT** have to do all three parts to register a mark for the assignment. You can stop doing this assessment item at any time, and whatever marks you have in PLATE at that time will be counted.

Minimum Requirements

To receive any marks, your solution must meet the following minimum requirements:

1. You must complete this assessment task in the order of the three parts, A, B, and C. You have to use your solution to Part A to do Part B, so you have no option but to do Part A before Part B. But you do NOT need to score full marks on Part A before doing Part B. You will need the full, correct functionality of Part A to do Part B, but you do not need any of the "design" and "indentation" marks to start on Part B. (The exact breakdown of marks for Part A is given later in this document.) Part C is a new program that does not require Parts A or B. However, **you must score at least 10 out of 15 on Parts A and B before you are eligible for any marks on Part C**.
2. Within each of Parts A, B, and C, the tasks must be implemented in the order specified in the "Task "sections below.
3. You may only use the features of Java that are taught in this subject. For example, you must not use inheritance, exceptions, varargs (e.g. printf), interfaces, or generics. We want to assess your ability to use the very specific features of Java that we have taught you in this subject.
4. Your solutions for Parts A and B must NOT use arrays (or equivalent).
5. Your program's output must EXACTLY match the output given by PLATE. To ensure you meet this requirement, it is highly recommended that you submit to PLATE frequently; at least once on each day that you do any work on the assignment.

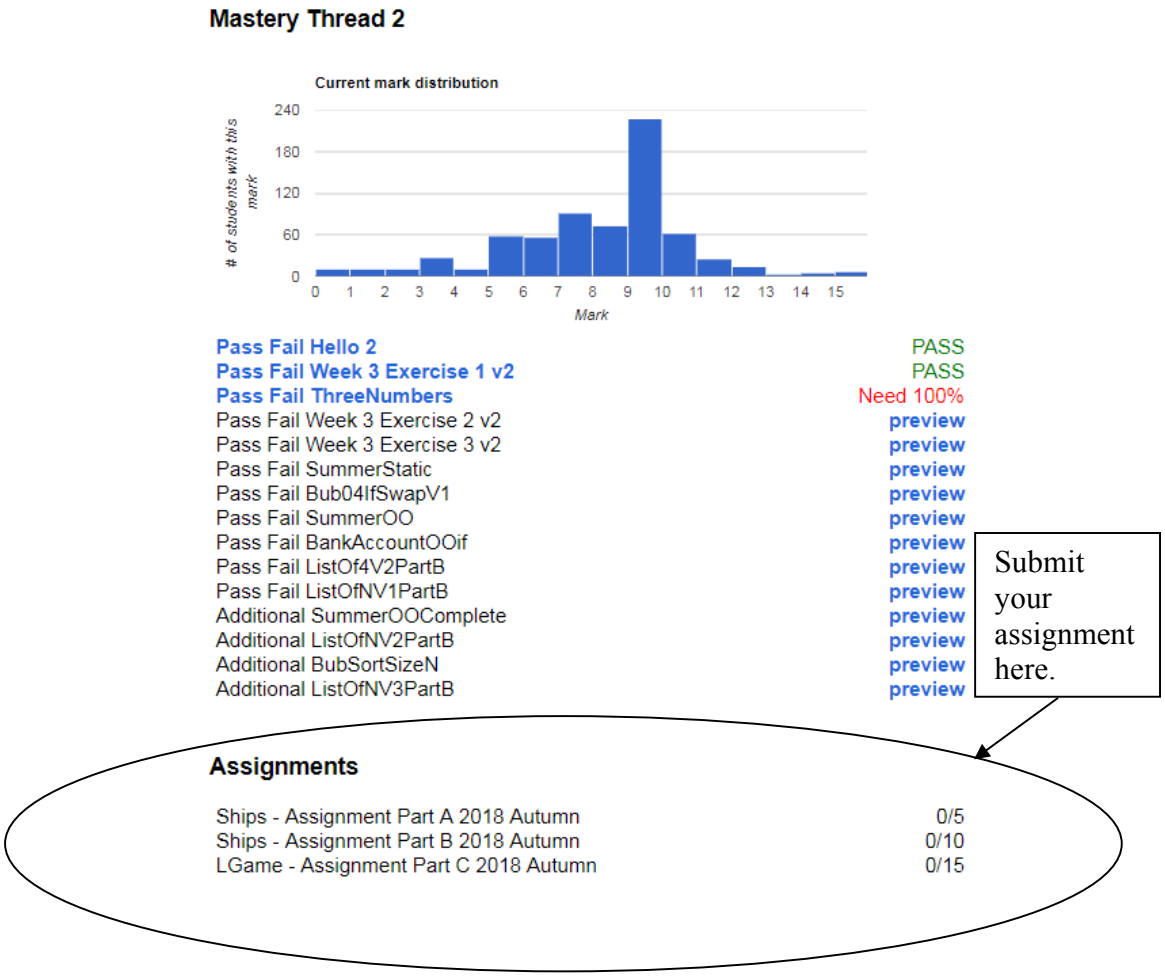
Assignment submission and return

Your assignment must be submitted as a JAR file through the PLATE system, online, at http://plate.it.uts.edu.au/. As shown in the diagram below, you submit your assignment on the same web page where you submit your lab tests; just lower down on that web page.

You may submit and resubmit as many times as you wish, before the due date, and PLATE will update your mark. Your mark for each part is available as soon as you submit to PLATE.

Further instructions for submitting to PLATE are displayed online at the PLATE website.

WARNING! PLATE may become overloaded on or near the due date when many students load and test their solution at the last minute. This will not be considered as a valid reason for an extension. To be safe, you should aim to submit your solution well before the due date.



NOTE: Unlike the lab tests, there is no "skeleton" file to download for the assignment.

Model solution

A model solution can be seen by contacting the subject coordinator via email **on or after Monday July 2.** This lengthy delay after the due date is because some students are likely to have been granted an extension for sickness, or misadventure (and such illness/misadventure has been documented). Do not email the subject coordinator to request the model solution before Monday July 2, as he will not be maintaining a list of names and thus your premature request may not be answered. Model solutions for each part of the assignment are only available to students who submitted that part of the assignment.

Academic misconduct (and submitting regularly)

See the above extract from the subject outline on plagiarism and academic integrity. To detect student misconduct, the subject uses an online system called PLATE available at http://plate.it.uts.edu.au

You must submit your progress to PLATE regularly while you are working on each task. This will provide us with a record that you have been doing your own work. If two students submit the same solution, your submission history may be used by the University Student Conduct Committee to determine who did the work and who copied.

This assignment is divided into separate tasks described below. You must submit your progress to plate regularly while attempting each individual task. That means **a student cannot submit one complete working solution at the end without any prior submissions to PLATE**. This will provide us with a record that you have been doing your own work.

If two students submit the same solution, your submission record may be used by the University Student Conduct Committee to determine which student did the work and which student copied. For more details on assignment submission, return and other important rules, scroll down to the last few pages.

Expected work load

It is expected that the workload for:

- **Parts A and B:** about 4 to 10 hours of work. A well-designed solution is expected to use approximately 150-200 lines of code, while a badly-designed solution may reach up to 300 lines of code. Some people may complete the task in 5 hours, and some may need 30 hours or more; there is a huge variation in students' experience and abilities.

- **Part C:** It is expected that this assignment will take about 15 to 20 hours of work. Some people may complete the task in 10 hours, and some may need 30 hours or more; there is a huge variation in students' experience and abilities. The model solution contains approximately 150 lines of code, excluding comments, blank lines, and lines containing only braces.

Seeking Help

Students should make the most of the many opportunities for face-to-face help in lectures and labs. Students are welcome to go to the **non-exam** hour of **ANY** lab session to seek help from a tutor. You do NOT have to be enrolled in a lab to get help from a tutor in the non-exam hour. **All tutors should be able to answer most questions about Part A of the assignment.**

Tutor Ahadi Alireza should be able to answer most questions about Parts A and B of the assignment. He has labs in every session except Wednesday, when Ryan Heise has a lab.

Tutor Ryan Heise should be able to answer most questions about all parts of the assignment, including Part C, having been the original author of Parts A, B, and C. The **non-exam** hours of Ryan's lab are:

- Wednesday, at **6pm** CB11.B1.400 (lab test in first hour is Wed17_B1_400_Ryan)
- Friday, at **5pm** CB11.06.102 (lab test in first hour is Fri16_06_102_Ryan)
- Friday, at **7pm** CB11.10.104 (lab test in first hour is Fri18_06_102_Ryan_2ndHourIn_10_104)

The subject coordinator may answer some simple questions by email that require a very short answer. However, if an emailed question would require a lengthy email reply, a student will be asked to seek help face-to-face, at either a lecture or lab session.

Students should NOT request help from tutors via email. They are only paid for their time in the lab. If yo email them, you are asking them to work for free ... do you work for free?

Should You Attempt the Assignment?

Students are reminded that they do NOT have to do the assignment to pass this subject. In fact, as the assignment is worth 30% of the subject, a student can score a credit in this subject without doing the assignment.

However, students who expect to follow this subject by doing Applications Programming (48024) are STRONGLY ENCOURAGED to do at least Part A of the assignment to prepare for Applications Programming. Of the students who did Programming Fundamentals in Autumn 2017 and who then went on to do Applications Programming (48024) in Spring 2017, **28%** failed Applications Programming. Of those students:

- Of the 33 students who scored **50/P** in Programming Fundamentals, **55%** failed Applications Programming.
- Of the 12 students who scored between **51 and 64 (i.e. a Pass)** in Programming Fundamentals, **33%** failed Applications Programming.
- Of the 20 students who scored **65 (i.e. the minimum mark for a Credit)** in Programming Fundamentals, **35%** failed Applications Programming.
- Of the 22 students who scored between **66 and 74 (i.e. a Credit)** in Programming Fundamentals, **27%** failed Applications Programming.
- Of the 24 students who scored between **75 and 84 (i.e. a Distinction)** in Programming Fundamentals, **8%** failed Applications Programming.
- Of the 34 students who scored between **85 and 100 (i.e. a High Distinction)** in Programming Fundamentals, **12%** failed Applications Programming.

Introduction to Parts A and B

It is expected that parts A and B will take about 4 to 10 hours of work. A well-designed solution is expected to use approximately 150-200 lines of code, while a badly-designed solution may reach up to 300 lines of code. Some people may complete the task in 5 hours, and some may need 30 hours or more; there is a huge variation in students' experience and abilities.

In parts A and B, you will create a simple Spaceship game with one player ship and three enemy ships. The player's ship can move left and right and can shoot enemy ships. The enemy ships are programmed to repeatedly move left and right. The game finishes when all enemies are destroyed. Sample output is shown below, with user input shown in **bold and underlined**:

```
Player position: 0
Enemy #1
- Initial position: 0
- Initial velocity: 1      .......x.....    ..........e..    ...........e.
Enemy #2                   ...e.........    ......e......    .........e...
- Initial position: -4     .......x.....    .....X.......    .....X.......
- Initial velocity: 1
Enemy #3
- Initial position: 2      _____xAx_____    ______iAi____    __________iAi_
- Initial velocity: -1     0 pts. move: f   1 pts. move: r   1 pts. move: f



......e......              ........e....    ...........e.    ..........e..
..e..........              ....e........    .......e.....    ..........e..
........e....              ......e......    .....X.......    .....X.......


_____iAi_____              _____xAx_____    _______iAi___    __________xAx_
0 pts. move: f             0 pts. move: f   1 pts. move: r   1 pts. move: f

(Output continues in
next column...)
                           .........e...    ............e    .........X...
                           .....x.......    ........e....    ...........X.
                           .....X.......    .....X.......    .....X.......


                           _____xAx_____    _________iAi__    __________xAx_
                           1 pts. move: r   1 pts. move: r   4 pts. You won!
```

Explanation of the sample output

The program begins by asking the user to input the initial position for the player ship, and the initial position and velocity for each enemy ship.

- **Initial position:** The left boundary of the coordinate system is at position -6, the right boundary is at position 6, and the middle of the screen is at position 0. Your program does **not** need to check if the user enters an initial position out of this range.

- **Initial velocity:** The user may enter either -1 or 1. Your program does **not** need to check if the user enters an invalid velocity. A velocity of -1 will cause the enemy to move left, while a velocity of 1 causes the enemy to move right.

The player's ship is placed at position 0. Then, the game enters a loop in which the current game state and score is printed to the screen, and the user is asked to enter a move. The move can be one of 3 options:

- **l**: move the player's ship one space to the left if it would not move out of bounds.
- **r**: move the player's ship one space to the right if it would not move out of bounds.
- **f**: fire both guns straight ahead, striking any and all ships in their paths. Given the power of the guns that is set (see below), 2 strikes are required to completely destroy an enemy ship. On the first strike, the ship will momentarily change from "e" to "x" and then back to "e". On the second strike, the ship will permanently change to "X" indicating that it has been destroyed.

Each time a move is performed, the enemies' moves happen before the player's own move happens. An enemy moves by moving left or right (depending on its velocity) until hitting a boundary, and then turns around to head in the opposite direction.

Each gun keeps track of its own points for each ship taken down, and the player's points are calculated as the total points from both guns. Each time an enemy is destroyed, 1 point is awarded to the gun that shot it, plus a bonus of 7-N points if $N \leq 7$ and if the previous ship was destroyed N moves ago by the same gun. (A move is counted as any of l/r/f.) Once all enemy ships have been destroyed, the game loop terminates, and the message "You won!" is printed.

Solution requirements

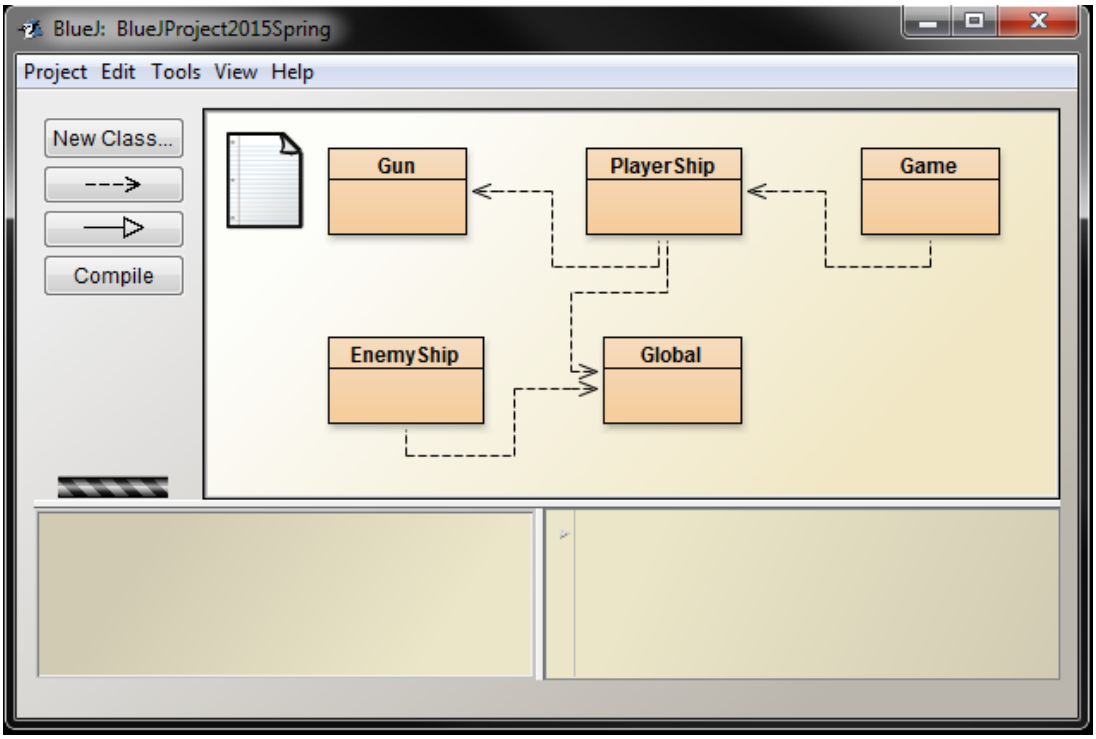
To receive any marks, your solution must meet the following minimum requirements:

- The tasks must be implemented in the order specified in section "Tasks" below.
- As a general rule, your solution must use only the features of Java that are taught in this subject. For example, students must not use inheritance, exceptions, varargs (e.g. printf), interfaces, or generics. Also, students must not use arrays (or equivalent, such as collections) in Parts A and B, even though arrays are taught in this subject.
- Your program's output must **exactly** match the output given by PLATE. White space (i.e. spaces, tabs and new lines) is significant in PLATE.
- You must define methods with the exact names and parameters as specified below (i.e. you must define methods with the given "*signatures*"), however you are permitted to define any number of additional methods of your own. Unless otherwise stated, you must not add additional fields to classes.

Part A (5%)

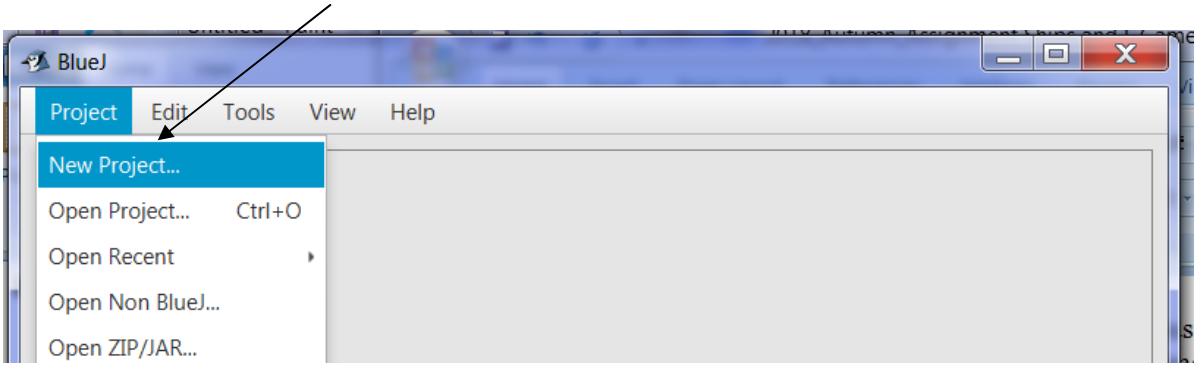
Students who expect to follow this subject by doing Applications Programming (48024) are STRONGLY ENCOURAGED to do at least Part A of the assignment to prepare for Applications Programming.

In this part, you will create the various classes that comprise the game. After completing part A, your project in BlueJ will look something like this:

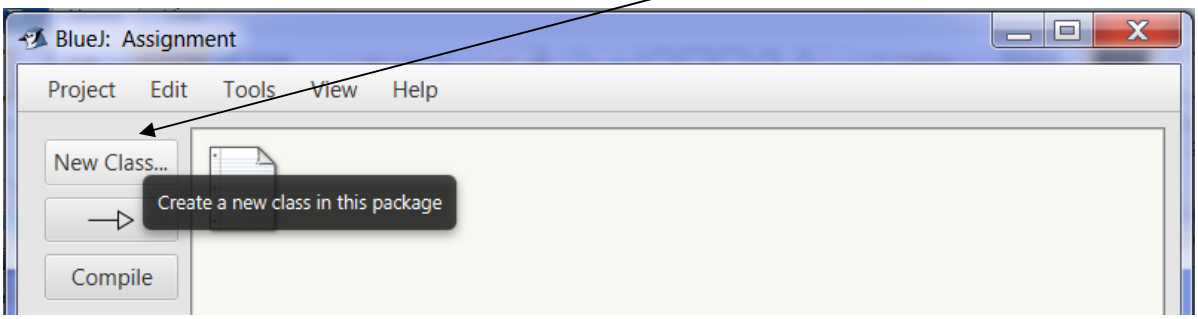


The actual position of your class icons will probably vary from what is shown above.

NOTE: Unlike the lab tests, there is no "skeleton" file for the assignment. To start the assignment, select "New project" off the "Project" menu, as shown below:



After creating the project, to create each new class, select “New Class” as shown below:



Do NOT save your JAR file to your BlueJ Project Folder

Do NOT save your JAR file into the same folder as your BlueJ files for the assignment. When you do that, each time you make a new JAR file, the new JAR contains the old JAR , and eventually you end up with a REALLY REALLY BIG JAR file. The JAR file for benchmark solution used in PLATE is only 5KB. Your JAR file should not be much bigger than that.

If you have already been saving the assignment JAR file into the same folder as the BlueJ files, then you'll need to:
(a) create a new BlueJ project folder,
(b) copy-and-paste your code from the old folder into the new folder, and then
(c) in future, don't save the jar file into this same new folder as the BlueJ files.

NOTE! THE ASSIGNMENT SUBMISSION SYSTEM RETAINS YOUR MOST RECENT ASSIGNMENT MARK, NOT YOUR HIGHEST ASSIGNMENT MARK. It is therefore possible to submit after the deadline and see your mark for the assignment go **DOWN**.

The coordinator will **NOT** be entertaining people who email to say that they "accidently" re-submtted the assignment after the deadline, or people who say "*I resubmitted just to find out what would happen if ...*". If you submit AFTER the deadline, and your mark goes down, then that lower mark will remain your mark for the assignment.

The Class Global

Rather than creating a new Scanner in several places in the Part A code, before starting Task 1 place the following code in the box below into a new class *Global*.

```
import java.util.Scanner;

public class Global
{
    // Rather than create a new Scanner in each of your classes,
    // use the global Scanner "keyboard" declared below

    public static final Scanner keyboard = new Scanner(System.in);


    /**
     * @return   an integer inputted by the user
     */
    public static int promptInt(String prompt)
    {
        System.out.print(prompt);
        int value = keyboard.nextInt();
        keyboard.nextLine();
        return value;
    }

    /**
     * @return   a move inputted by the user
     */
    public static char readMove()
    {
        System.out.print("move: ");
        char move = keyboard.nextLine().charAt(0);
        return move;
    }
}
```

Because the "keyboard" field has been defined as "public static", it can be accessed from any other class in the program.

Technical note: In *Global*, methods "promptInt" and "readMove" have been provided to avoid a complication of using the *Scanner* built-in method "nextInt". Method nextInt() will read an integer from the keyboard, but it will **NOT** also read the newline character (i.e. when the user presses the enter key). This is different behaviour from the way method nextLine() works, which reads everything on a line **including** the newline character. Notice how in the method "promptInt" of class *Global,* the line containing a call to "nextInt" is followed on the next line by a call to "nextLine", to get rid of the newline character. **In general, whenever you need to read an integer, or a move command, use the methods "promptInt" and "readMove" in class *Global*.** If you need to use "nextInt" to do some other form of input (and you may need to in your **Part B** code, but NOT Part A), to avoid any problems after using nextInt(), you should always immediately consume the remaining newline character by adding this line:

```
Global.keyboard.nextLine();
```

Tasks

These tasks must be completed in the order listed below. **As you complete each Task, submit your code to PLATE, to establish a record of steady work on the assignment.**

Task 1: Classes and fields

Create the following *classes* (written in italics) and "fields" / "private data members" (enclosed in double quotes):

- Each *Gun* has a "position", "power" and "points", all integers. Each gun also has a "justFired" status which can be either true or false (i.e. of type **boolean**; see the free textbook online by Parsons, section 3.1.3, "Boolean Variables", page 33, or google ... `java tutorial boolean`).
- The *PlayerShip* has "position", an integer, and two instances of *Gun*, called "gun1" and "gun2".
- Each *EnemyShip* has "position", "velocity" and "life", all integers. Each enemy ship also has a "justHit" status which can be either true or false (i.e. of type **boolean**; see the free online textbook by Parsons, section 3.1.3, "Boolean Variables", page 33). The class *EnemyShip* also maintains a count of the number of enemy ships that have been created so far, called "number", an integer, which is not attached to any one particular enemy ship, but rather is associated with the whole class of enemy ships. Thus "number" in *EnemyShip* has a similar role to "numSummers" in *SummerOO*.
- A *Game* has 1 *PlayerShip* called "player", and 3 instances of *EnemyShip* called "enemy1", "enemy2" and "enemy3".

In the above bullet point descriptions, note that the names of the classes and fields follow Java's naming conventions. That is, class names begin with an uppercase letter, field names begin with a lowercase letter, compound nouns are joined without a space and the second word begins with an uppercase letter).

Note: all fields should be declared as `private`.

When you have completed this entire task, submit your code to PLATE.

Task 2: Add Constructors to the Classes

2.1 Gun

A constructor for class *Gun* takes two integer parameters. The first parameter is the initial value for the field "position". The second parameter is the initial value for the field "power".

2.2 PlayerShip

Using the "promptInt" method in class *Global,* a constructor for class *PlayerShip* reads input from the user and initialises the "position" field. Your code should provide the String "`Player position: `" as a parameter to method "promptInt". (Note that in the String there is a space after the colon.) The following is an example of what this I/O looks like when the constructor executes (the **bold and underlined** text indicates the user's input):

```
Player position: 3
```

The PlayerShip constructor also creates two guns with power 5, with the first gun immediately one position to the left of the player ship, and the second gun immediately one position to the right of the player ship.

2.3 EnemyShip

A constructor for class PlayerShip reads input from the user and initialises the position and velocity of the ship according to the following I/O (**bold and underlined** text indicates sample user input, and again note the space after the colon):

```
Enemy #1
- Initial position: 3
- Initial velocity: -1
```

The enemy ship's life is initialised to 10, and the "justHit" status is initialised by default to false.

The constructor increments the variable "`number` "which records the number of enemy ships. Java by default initialised this field to 0. Thus, when the first enemy is created, `number` increments to 1. When the second enemy is created, `number` increments to 2, and so on. In the first line of the example above, "`Enemy #1`", the "1" is the value in that variable "`number".`

2.4 Game

A constructor for class Game creates the player ship and three enemy ships.

When you have completed this entire task, submit your code to PLATE.

Task 3: The move() method

The class PlayerShip has a method called move() that takes an integer as a parameter, which can be positive or negative. When the move () method is invoked, that parameter value is added to the player ship's current position. A negative distance will have the effect of making the ship move left. Note that this method should be written *compositionally* so that, when the ship is moved, not only is the field "position" in the instance of *PlayerShip* updated, but:

- The ship's guns also move in the same direction and by the same distance.
- A method "move" should also be written in the Gun class to update the value for the field "position" in that class.
- The "move" method in the PlayerShip class should make two calls (one call for each gun) to the "move" method in the Gun class.

When you have completed this entire task, submit your code to PLATE.

Task 4: toString() method

The class Game contains a toString() function that returns (but does not print) a string of the form:

```
Enemy(-4) Enemy(-2) Enemy(5) Player[2, 1pts]
```

The above string indicates that the first enemy is at position -4, the second is at position -2, the third enemy is at position 5, and the player is now at position 2 with currently 1 point scored. To calculate the number of points scored for the player, simply add together the number of points scored by each gun.

This functionality is implemented *compositionally*. That is, Game.toString() does its job by calling the toString() functions in other classes to do some of the work:

- EnemyShip.toString() returns a string like, for example, "Enemy(2)".
- PlayerShip.toString() returns a string like, for example, "Player[3, 4pts]"

When you have completed this entire task, submit your code to PLATE.

Marking scheme for Part A

Your solution will be marked according to the following scheme:

Task 1: Classes and fields	20 marks
Task 2: Constructors	50 marks
Task 3: move() method	10 marks
Task 4: toString() method	20 marks
Correct indentation penalty	Up to 15 marks deducted

Correct indentation means that code should be shifted right by one tab between { and }.

Note: PLATE shows marks out of 100. That PLATE mark will be converted to a mark out of 5. When making that conversion, the mark out of 5 will be rounded to the nearest integer. **Thus a mark of 90 out of 100 is 4.5 out of 5, which will be rounded to 5. There is therefore no point to trying to achieve a mark higher than 90 out of 100.**

Note: You have to use your solution to Part A to do Part B, so you have to do Part A first. But you do NOT need to score full marks on Part A before doing Part B. You need PLATE to award you a "PASS" for Tasks 1-5, but you do not need the "indentation" marks to start on Part B.

Part B (10%)

In Part B of the assignment, you will finish the game that you started building in Part A, using your solution to Part A as a starting point.

As you complete **each and every task below**, submit your code to PLATE to receive marks for that part**,** to establish a record of steady work on the assignment.

Note!! Your Part B solution should be submitted on PLATE under the link "Assignment Part B". Be careful **NOT** to submit under the link "Assignment Part A".

Tasks

Task 1: Move player

Define a method called movePlayer() in class *Game* that reads and executes the next move of the player according to the following I/O (**bold** text indicates sample user input):

```
move: r
```

To do this, use the method "readMove" in class Global.

We recommend that you use a "switch" statement in this task. Here is an example of processing a menu with a switch statement:

```
System.out.print("Would you like to quit? (Y/N) ");
String line = keyboard.nextLine();
char answer = line.charAt(0);

switch (answer)
{
    case 'y':
    case 'Y':
        System.out.println("Bye.");
        break;

    case 'n':
    case 'N':
        System.out.println("Excellent!");
        break;

    default:
        System.out.println("Invalid answer.");
        break;
}
```

You can find an explanation of switch statements in the free online textbooks:

- Nielsen: section 2.2.5, "Multiple choices: switch case", pages 39-40.
- Parsons: section 4.1.6 "switch Statements", pages 56-58.

Note: PLATE only marks your code on the basis of the output generated by your code, so you are NOT required to use a switch statement.

If the character entered was 'l' or 'L', then make the player's ship move left. If the character entered was 'r' or 'R', then make the player's ship move right. The player ship can only move within the bounds -6 to 6 and any attempt to move out of bounds should have no effect. Also keep in mind that the ship has a gun on either side, and the guns also cannot move out of bounds. Your out-of-bounds check should be implemented ***compositionally***. That is, rather than writing all code in class Game, implement a "move" method inside class *PlayerShip*, which is called by method movePlayer in class *Game* to do some of the work. Furthermore, implement a "move" method inside class *PlayerShip* , which is called by move method in class *PlayerShip* to do some of the work.

Task 2: Move enemies

Define a method called moveEnemies() in class *Game* that moves each enemy ship one step by adding its current velocity to its current position. For example, if a ship is currently at position 3 with velocity -1, it should move to position 2.

If an enemy ship hits a boundary, it should turn around and move in the opposite direction. This can be achieved by flipping the ship's velocity from a negative into a positive number, or from a positive into a negative number. The behaviour should be as follows:

- If the ship is at position -5 at velocity -1, moveEnemies() moves the ship to position -6
- If the ship is at position -6 at velocity -1, moveEnemies() will flip the velocity to 1, and then move the ship to position 5.

This method moveEnemies() in class *Game* should be implemented ***compositionally***. That is, this method should call a method "move" in the class *EnemyShip* to do some of the work.

Task 3: Fire weapons

Modify the movePlayer() method so that if the user types in 'f' or 'F' after being prompted to enter a move (i.e. instead of typing 'l', 'L', 'r' or 'R' as implemented before) then the player's ship will fire both guns which will strike any and all ships in the path of each gun. If an enemy ship is struck, its life should be decreased by the amount of power in the gun that shot it. Do not assume that the power of the guns is always 5 as PLATE may test your guns with different power settings. If a ship is destroyed (i.e. its life is decreased to zero), then points should be awarded as described in the earlier Section "*Explanation of the sample output*". To implement this task, you are permitted to add ONE additional field to your program. Again, this functionality is probably most easily implemented ***compositionally***. That is, the movePlayer() method should call a method in the class *Player* to do some of the work, and in turn that method in Player should call a method in the class *Gun* to do some of the work

Task 4: Larger velocities 

Modify your program so that the user can enter larger velocities than 1 and -1. For example, if an enemy ship is currently at position -2 and its current velocity is 5, it should move to position 3.

Note: Your method should respect boundaries. For example, if an enemy ship is at position 3 with velocity 5, the ship will move 3 steps to the border, the velocity will be flipped to -5, and the ship will continue the remaining 2 steps in the opposite direction, landing the ship finally at position 4 with velocity -5.

If you find it difficult to formulate a solution, try simulating the above example through role playing:

- Pretend that you are the enemy ship, and that you are at position 3 with velocity 5.
- Now move yourself to where the enemy ship should move to.
- Did you succeed? *How* did you decide where to move to? Write down the decision process you used in your own thoughts, and translate this into your computer program.

This task is not a prerequisite for later tasks, so you may (if you prefer) try Task 5 first and come back to Task 4 later.

Task 5: Printing

Defined a method called print() in class Game that prints a representation of the game in its current state, exactly according to the following output:

```
Normal output                          Output after bullets fired



..........e.                           .........x...
.........e...                          ..........x.
.....e.......                          .....e.......


_______iAi___                          _________xAx_
1 pts.                                 3 pts.
```

The output consists of 3 blank lines followed by 3 lines of enemies (enemy #1 is first) followed by two more blank lines followed by a line displaying the player's ship with its guns followed by a printout of the current number of points scored. The final line is a string of the form "`1 pts. `" **where a single "space" is included after the full stop**. This final line is **not** terminated by a new line character. That is, you should use println() to print all lines except for the last line which should be printed using print().

You might find this task very challenging unless you break down the problem into smaller parts using a compositional design. That is, define a print() method in each class to perform part of the overall goal:

- EnemyShip.print() should print one row such as "`.........e...`".
- PlayerShip.print() should print one row such as "`_______iAi___`".
- Gun.print() should print one character such as "`i`" or "`x`".

Once again, you may find it difficult to formulate a solution for each print() method. It is suggested again that you try to simulate the job of printing one row using role playing. Pretend to be the computer, and attempt to print one row, one character at a time. Ask yourself, ***how*** did you decide what character to print? Write down your decision process and translate this into your computer program.

Task 5: Main program

Define a class called Main and within it a standard `public static void main(String [] args)` method which triggers the entire game to play exactly according to the sample output at the top of this document.

Marking scheme for Part B

Your solution will be marked according to the following scheme:

Task 1: Move player	10 marks
Task 2: Move enemies	15 marks
Task 3: Fire weapons	17 marks
Task 4: Larger velocities	10 marks
Task 5: Printing	15 marks
Task 6: Main program	3 marks
Correct indentation	5 marks
Design	25 marks

Correct indentation means that code should be shifted right by one tab between { and } and within nested control structures such as if statements, switch statements and loop statements. In a switch statement, the code within each case should also be indented. NOTE: You are responsible for your own indentation. Do not rely on BlueJ to indent code for you. BlueJ does not always indent code in the way that is required.

Design marks are awarded for the quality of your solution. More marks are awarded for placing code into the most appropriate classes so as to increase *cohesion* and reduce *coupling*. (To understand coupling and cohesion better, google the words … java tutorial coupling cohesion.)

Note: PLATE shows marks out of 100. That PLATE mark will be converted to a mark out of 10. When making that conversion, the mark out of 10 will be rounded to the nearest integer. **Thus a mark of 95 out of 100 is 9.5 out of 10, which will be rounded to 10. There is therefore no point to trying to achieve a mark higher than 95 out of 100.** *Students who wish to achieve a High Distinction, but who struggle to collect sufficient indentation and design marks to get their mark for Part B up to 95, are advised to work on Part C, rather than try to eek out those last few marks from Part B.*

Part C (15%)

Note: *you must score at least 10 out of 15 on Parts A and B before you are eligible for any marks on Part C.*

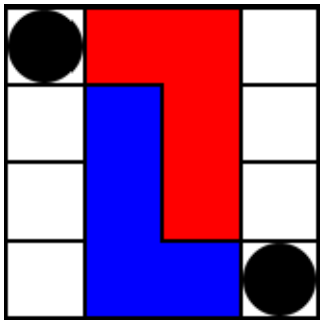
Note: *Only students hoping to achieve a High Distinction should attempt Part C.*

The goal of Part C is to create a simple game using arrays called the L game.

The following description of the rules of the game is from Wikipedia.

http://en.wikipedia.org/wiki/L_game

Description



The L game is a two-player game played on a board of 4×4 squares. Each player has a 3×2 L-shaped piece, and there are two 1×1 neutral pieces (black discs in the diagram).

Rules

On each turn, a player must first move their L piece, and then may optionally move one of the neutral pieces. The game is won by leaving the opponent unable to move his L piece to a new position.

Pieces may not overlap or cover other pieces. On moving the L piece, it is picked up and then placed in empty squares anywhere on the board. It may be rotated or even flipped over in doing so; the only rule is that it must end in a different position from the position it started—thus covering at least one square it did not previously cover. To move a neutral piece, a player simply picks it up then places it in an empty square anywhere on the board.

You will implement this game by printing the board to System.out (as ASCII characters), and read moves from the user via System.in (using a Scanner). Sample output will be given in the task descriptions below.

1. Minimum requirements

To receive any marks, your solution must meet the following minimum requirements:

- You must regularly submit to PLATE while developing your solution. You should definitely NOT suddenly submit a completely working solution on the due date without submitting your progress, because there will be no evidence that you developed your solution. Any student who submits a substantial solution without an equally substantial record of previous submissions during development will receive ZERO.

- Individual tasks will not receive marks unless your solution's output exactly matches the benchmark solution's output that is shown when submitting to PLATE.

- PLATE will require you to complete some tasks before continuing on to later tasks. Therefore, you will need to submit frequently to PLATE to see whether PLATE allows you to skip over the current task, or whether it forces you to complete the task before continuing.

2. Specification and Tasks

Task 1: Slide (50 marks)

While there are many ways to implement this game, in this assignment, the game should be conceived of as a set of 4 moving "slides", each slide depicting a different game piece within a 4x4 area which is to be represented by a 2D array. For example, a slide representing the blue L piece in the above picture might be represented by the following array:

```
{
    { ' ',' ',' ',' ' },
    { ' ','o',' ',' ' },
    { ' ','o',' ',' ' },
    { ' ','o','o',' ' }
}
```

In this array, the colour blue is represented by the character 'o', and empty space is represented by the ASCII space character.

Each slide has the ability to be moved around, rotated or flipped (but for now we will just focus on moving). In this task, you need to define class Slide with the following structure:

```
public class Slide
{
    private char[][] cells;
    public Slide() …
    public Slide(char[][] cells) …
    public void print() …
    public void clear() …
    public void project(Slide other) …
    public void move(int row, int col) …
}
```

The first constructor should initialise field "cells" with a new blank 4x4 array.

The second constructor should initialise field "cells" with the parameter cells.

The print() method should print each row of the cells on a separate line (row 0 at the top). Each cell in the same row should be printed with a space after each cell. Note that each cell in the array should be accessed by cells[rowPosition][colPosition].

The clear() method should set each of the cells to an ASCII space character.

The project(other) method should take another slide as a parameter, and copy each non-space cell from this slide into the other slide (i.e. like a projector projecting slides onto a screen).

The move(row, col) method should shift the contents of the array vertically and horizontally so that the top-left visible corner of the piece is at cells[row][col].

Here is a simple test that you can try in the BlueJ code pad:

```
Slide slide = new Slide(new char[][] {
        { 'x','x',' ',' ' },
        { 'x','x',' ',' ' },
        { ' ',' ',' ',' ' },
        { ' ',' ',' ',' ' }
    });
slide.print();
```

(You should now see the following output)

```
x x
x x
```

Now type this in the code pad:

```
slide.move(2, 1);
slide.print();
```

(You should now see the following output)

```
  x x
  x x
```

You can assume that the move() method is always given valid parameters such that the piece will never be moved outside of the 4x4 box.

If you find the move() method too difficult, you are allowed to request the solution by sacrificing 10 marks. This might be a reasonable sacrifice for students who struggle with this task for many hours. Requests for the solution should be emailed to Ryan.Heise@uts.edu.au and CC Suresh.Paryani@uts.edu.au . Note that this solution must not be shared with anyone as this would be classified as academic misconduct (refer to the section “Academic Misconduct” below).

Task 2: LGame (20 marks)

In this task, you will write the game program. The entry point to your program must be a class called LGame and this class must define a standard main() method. You should create 4 slides representing the 4 pieces arranged in the following relative positions using the following character symbols:

```
A i i
  o i
  o i
  o o B
```

Each slide should contain only the characters representing one piece. The two neutral slides are represented by the character symbols 'A' and 'B' while the two L-piece slides use the symbols 'o' and 'i'.

Create a 5th slide called "screen". Since each of the 4 pieces are represented by 4 different slides, in order to print everything onto the screen, you should first clear the screen, project each of the 4 slides onto the screen, and then print the screen.

Your game should involve a loop where you read the next move from the player, and then perform the move, and then print the changed slides. The user should enter a move in the following format:

Move: o12

Here, the user types in a string with length 3 characters. The first character indicates which piece/slide the user wants to move. This can be 'o' or 'i' or 'A' or 'B' representing the symbols for each piece described above. The second and third characters represent the row and column position to move the piece to respectively.

When your program is run via its main method, it should match the following input/output format:

```
A i i
  o i
  o i
  o o B
Move: o10
A i i
o   i
o   i
o o   B
Move: A11
  i i
o A i
o   i
o o   B
Move: B00
B i i
o A i
```

```
o   i
o o
Move: i12
B
o A i i
o     i
o o   i
Move: end
Game over
```

3.

Your program does not need to detect a winning position. You simply end the program when the user types "end", and it will be the user's responsibility to follow the game rules.

Task 3: Improvements (30 marks)

Improve your program in the following ways:

- If the user attempts to move a piece to a position on the 4x4 board that is already occupied, print "Invalid move" and do not perform the move.
- If the user inputs a move with a 4th character, the 4th character indicates whether the piece should be flipped over from left to right. For example, the move "i001" means flip the L-piece with symbol 'i' and move it to row 0 and column 0. The move "i000" does no flip and has the same effect as "i00".
- If the user inputs a move with a 5th character, the 5th character is interpreted as an amount to rotate the piece. 0 means don't rotate. 1 means rotate 90 degrees clockwise. 2 means rotate 2*90 degrees clockwise and 3 means rotate 3*90 degrees clockwise. Obviously 4 has the same effect as 0. For example, the move "o1102" will rotate the piece 180 degrees and move it into row 1, column 1. It is possible to specify both a flip and a rotation in the same move, and both should be performed.

Your program should match the input/output of the following example:

```
A i i
  o i
  o i
  o o B
Move: i00
Invalid move
A i i
  o i
  o i
  o o B
Move: i02
A   i i
  o   i
```

```
  o   i
  o o B
Move: o101
A   i i
  o   i
  o   i
o o   B
Move: o100
A   i i
  o   i
  o   i
o o   B
Move: o1002
A   i i
o o   i
o     i
o     B
Move: B01
A B i i
o o   i
o     i
o
Move: i2113
A B
o o
o i
o i i i
Move: A03
  B   A
o o
o i
o i i i
Move: end
Game over
```

You are not required to check that the user is taking turns appropriately; It is up to the players of the game to take turns moving their own piece.

Marking scheme for Part C

Your solution will be marked according to the following scheme:

Task 1: Slide	(50%)
Task 2: LGame	(20%)
Task 3: Improvements	(30%)
Indentation consistency	1% is subtracted from your total for each indentation mistake. To indent properly, code should be shifted right by one tab between { and } and within nested control structures such as if statements, switch statements and loop statements. In a switch statement, the code within each case should also be indented.

Note: PLATE shows marks out of 100. That PLATE mark will be converted to a mark out of 15. When making that conversion, the mark out of 15 will be rounded to the nearest integer. **Thus a mark of 97 out of 100 is 14.5 out of 15, which will be rounded to 15. There is therefore no point to trying to achieve a mark higher than 97 out of 100**.