

---

# Workbook 3

## Introduction

In this workbook you will use your implementation of `PackedLong` from last week as a data structure in a simple implementation of Conway's Game of Life. Using this data structure imposes severe constraints on the size of game board which can be simulated. Next week you will overcome this limitation by making use of a Java array to store a game board of an arbitrary size.

### Important

Make sure you have collected a questionnaire about the previous week's work. You should complete it and hand it to your tucker as they come round.

Please replace `crsid` with your own username in all examples and instructions.

Remember to check the course website regularly for announcements and errata:

<http://www.cl.cam.ac.uk/teaching/0809/ProgJava>

If you finish this exercise early, you may also wish to check the website for a list of additional projects.

## Conway's Game of Life

John Conway was an undergraduate at Cambridge and read Mathematics. He stayed on at Cambridge to study for a Ph.D. and afterwards as a Lecturer. Conway invented the *Game of Life* in 1970. You can play the Game of Life with physical game pieces (a set of *stones* from Go<sup>1</sup> are a good choice) but a computer simulation of the Game of Life allows for quicker experimentation.

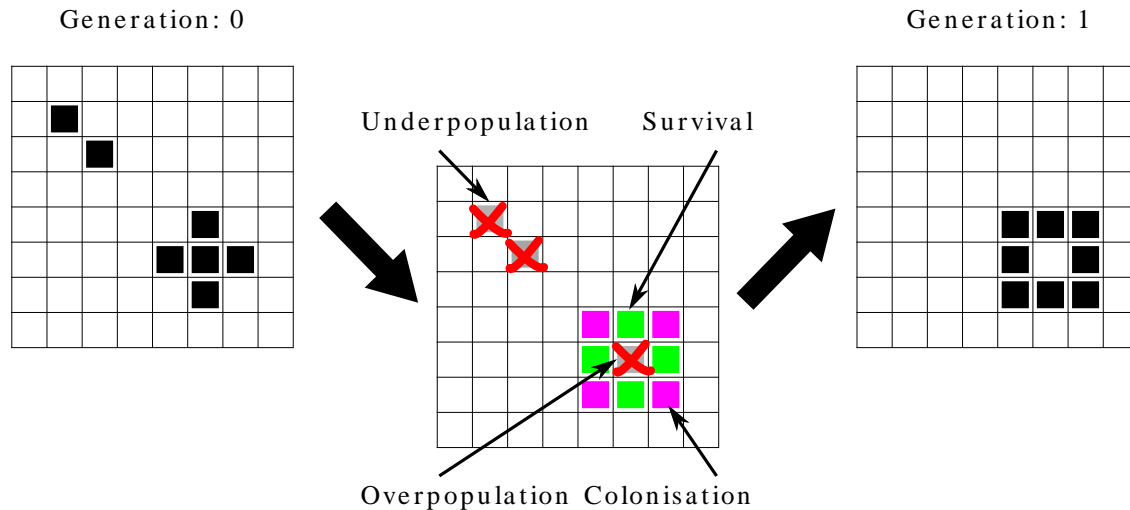
The game board, or *world*, for the Game of Life is a two-dimensional grid of square cells. Each cell in the world is in one of two states, *dead* or *alive*. The world transitions through a set of discrete *generations*, starting from the initial state of the cells at time zero, which is determined by the human player. The rules of the game describe how to transition from generation  $t$  to generation  $t+1$ , and are as follows:

- a live cell with fewer than two neighbours dies (caused by underpopulation);
- a live cell with two or three neighbours lives (representing a balanced population);
- a live cell with with more than three neighbours dies (caused by overcrowding and starvation); and
- a dead cell with three live neighbours comes alive (colonisation).

The game calls for these rules to be applied simultaneously to all cells in order to produce the next generation i.e. a cell which dies due to underpopulation could also play a role in colonisation of another cell. We will implement this by applying the rules to each cell in turn and writing the results into a new, blank, world rather than updating the current one. An example showing the application of the rules is given in Figure 1, "Applying the Game of Life rules".

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Go\\_\(board\\_game\)](http://en.wikipedia.org/wiki/Go_(board_game))



**Figure 1. Applying the Game of Life rules**

## Outline

In this workbook you will build an implementation of Conway's Game of Life using the `PackedLong` class you wrote last week. The `PackedLong` class is capable of storing at most 64 `boolean` values in a variable of type `long`. We will use these `boolean` values to represent the liveness of a cell; in other words, if the cell is alive then `true` is stored, and if the cell is dead then `false` is recorded. The limit of 64 `boolean` values restricts the size of the game board to an eight-by-eight world. Given this size restriction, this particular implementation of Conway's Game of Life is called *TinyLife* in this workbook.

You may recall that, given a variable `v` of type `long`, the `set` method of `PackedLong` class is able to set bit `i` of `v` to the value `val` with the method call:

```
v = PackedLong.set(v, i, val)
```

Note in particular the use of the assignment operator to update the value stored in the variable `v`. Similarly, the state of bit `i` of the variable `v` of type `long` can be retrieved and stored in variable `b` of type `boolean` using the method call:

```
b = PackedLong.get(v, i)
```

To use `PackedLong` to store the the state of the world, a mapping from cell location on the game board to the bit location in the variable is required. In order to maintain compatibility with the examples and test cases in this workbook you must use the mapping described here, although in principle there are 64 factorial possible valid mappings to choose from (most of which would be more tedious to implement than the one used here).

In this workbook you should use the bits stored by the `PackedLong` class so that the state of top left cell of the board is stored in the least significant bit, the bottom right cell of the board is stored in the most significant bit, with the cells in between stored, in increasing bit positions, in row order. Figure 2, "Bit positions in `PackedLong` used to store the state of cells in *TinyLife*" contains a graphical illustration of this specification.

0	1	2	3	4	5	6	7
8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23
24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55
56	57	58	59	60	61	62	63

**Figure 2. Bit positions in `PackedLong` used to store the state of cells in `TinyLife`**

We will refer to an individual cell in the Game of Life by its column,  $c$  and row,  $r$  in the grid, written as a pair:  $(c,r)$ . The cell  $(0,0)$  is at the very top left of the world, and the cell  $(7,7)$  is at the bottom right of the world. For example, in Figure 2, “Bit positions in `PackedLong` used to store the state of cells in `TinyLife`”, the state of cell  $(0,0)$  is recorded at bit position 0, and the state of cell  $(3,2)$  is recorded at bit 19.

#### Converting cell row and column index to a `PackedLong` index.

Write answers to all of the questions found in this workbook into a plain text file called `answers.txt`. You will need to include your `answers.txt` file in your submission for your third Java Tick. Further information about the submission of the tick can be found at the end of this workbook.

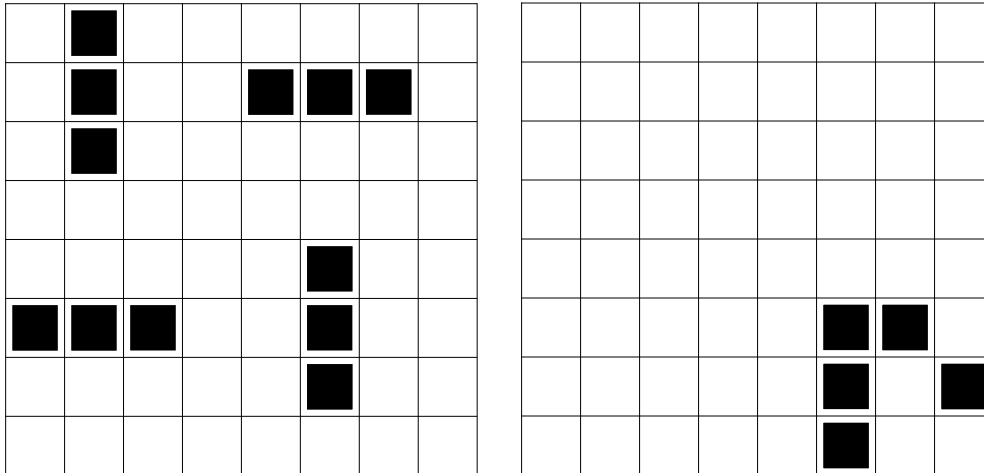
A Java programmer has used the variables  $x$  and  $y$  to record the location of the cell  $(x,y)$ .

1. Write a Java expression which uses the variables  $x$  and  $y$  to determine the bit position used to store the state of cell  $(x,y)$ . For example, if  $x$  equals 1 and  $y$  equals 2, then your expression should return 17.
2. If the state of a cell is stored at bit position  $i$ , write a Java expression to determine the value of  $x$ , the column in the game board which contains the cell. For example, given the value 17 your expression should return the value 1.
3. If the state of a cell is stored at bit position  $i$ , write a Java expression to determine the value of  $y$ , the row in the game board which contains the cell. For example, given the value 17 your expression should return the value 2.

## Initialising the world

Initialising the world in `TinyLife` is easy: simply define a new variable of type `long` which contains the bit pattern representing the state of each of the 64 cells in the world. For example, the definitions of the variables `oscillator` and `glider` below contain the correct bit patterns to represent the worlds shown in Figure 3, “Example `TinyLife` worlds”. The printed workbook depicts the state of these worlds at generation zero; the on-line version of this workbook contains animated graphics which depict how the state of the world evolves over time. (These graphics are animated GIF files; you will learn how to build your own such animated graphics in Tick 4\*.)

```
long oscillator = 0x20272000027202L;
long glider = 0x20A0600000000000L;
```



Left, a *two phase oscillator* (code: 0x20272000027202L) and right, a *glider* (code: 0x20A0600000000000L).

**Figure 3. Example TinyLife worlds**

A student defines a world as follows:

```
long world = 0x2623321111233262L;
```

4. What is the value of the bit found in position 6 in the variable `world`?
5. What is the value of cell (4,2) in the variable `world`?

## Retrieving and updating cells

As you proceed through this workbook, you will fill in key parts of a program which will draw the state of TinyLife as textual characters to your terminal. The program that you will create will form your submission for Tick 3. Open a text editor and create a new class called `TinyLife` inside the package `uk.ac.cam.crsid.tick3`. Add a special "main" method to the class `TinyLife`. Check that your skeleton class works by inserting a Java statement to print out your name and compile and run your program.

Copy the file `PackedLong.java`, which you submitted as part of your work for last week, into the same directory as `TinyLife.java`. You will need a copy of `PackedLong.java` in the same directory as `TinyLife.java` so that your modifications to `TinyLife` can make use of the features provided by `PackedLong`. If you have not yet completed your implementation of `PackedLong` you will need to complete this first before continuing with this workbook. Remember to change the package of the `PackedLong` class to reflect its new location. You should not leave the `PackedLong` class in the `tick2` package and call it from the `tick3` package. If you did this it would make your tick submission difficult because your tick submission will rely on the previous week's submission.

In this section you will write two *methods* which will be used later in your implementation of `TinyLife`. A method is a unit of computation associated with a class in Java. A method provides a well-defined interface, specifying the values required to perform the computation described by the method, and the

type of the result returned (if any). You saw two example methods last week when you wrote your implementation of `PackedLong`; the methods were called `set` and `get`.

A method has a prototype which describes the types of the arguments which will be provided when the method is invoked, and the type of the result returned when the execution of the method finishes. The prototypes of the two methods you will write are in this section are:

- `public static boolean getCell(long world, int x, int y)`
- `public static long setCell(long world, int x, int y, boolean value)`

A method also has a *body*, which is a sequence of statements describing the computation the method should perform. The body of a Java method is encapsulated with opening and closing curly brackets (`{` and `}`). In Java the method prototype and body form a *method definition* and must be placed directly inside a Java class; you cannot place a method definition inside another method in Java.

## Conditional execution

When writing the body of a method, it is a good idea to check the values provided to the method to make sure they are reasonable. This is sometimes called *input sanitisation*. For example, the method `getCell` accepts three arguments. All possible values of type `long` for the `world` argument of `getCell` are valid descriptions of a world in `TinyLife`, therefore no input sanitisation is required for this variable. The values of the second and third arguments, `x` and `y`, are only valid in the range from zero to seven; and input sanitisation must ensure that the method handles the case when the values of either `x` or `y` are outside this valid range. In the case of `getCell`, an acceptable solution is to return the value `false` from the method if `x` or `y` are greater than seven or smaller than zero. (This is equivalent to saying cells which are outside the 8-by-8 world are always dead.)

To do this input sanitisation we need to check the value of `x` or `y` and execute a different piece of Java program if the value of the variable is outside the valid range. In Java, conditional execution is performed with an `if` statement. You saw an example `if` statement in the body of the method `PackedLong.set` in Tick 2. An `if` statement is written as follows in Java:

```
if (boolean_expression)
    statement1
else
    statement2
```

You can replace `statement1` or `statement2` with a Java statement (e.g. `"i = 2 * 6;"`). If you want to execute more than one Java statement conditionally, then you can use curly brackets to group together a set of Java statements into a *basic block* and use the basic block in place of `statement1` or `statement2`. The `else` clause and `statement2` can be removed if you only want to execute statements if `boolean_expression` evaluates to `true`. Here are some examples:

```
if (true)
    i=1;

if (i == 1) {
    System.out.println("true");
    i = 0;
} else {
    i = 1;
}
```

An alternative form of conditional execution operates *within* a Java expression, and has the form:

```
boolean_expression ? expression1 : expression2
```

Here, if `boolean_expression` evaluates to `true` then `expression1` is evaluated; otherwise `expression2` is evaluated. A typical usage of this latter form of conditional execution is:

```
int i = boolean_expression ? one_var : another_var;
```

in which the value of `i` is initialised with the current value stored in `one_var` if `boolean_expression` evaluates to `true`, or with the current value stored in the variable `another_var` if `boolean_expression` evaluates to `false`.

Your next task is to use conditional execution to write an implementation of `getCell`. Your method's prototype should match the one given above. If either `x` or `y` are out of range it should return `false`, otherwise it should work out the corresponding bit-position, call the `PackedLong.get` method and return the result.

You should test your code by updating your copy of `getCell` in `TinyLife.java` and placing a temporary piece of test code at the end of the `main` method. The temporary code should invoke `getCell` with the values of `world`, `x` and `y` as shown in Table 1, "Test patterns for the `getCell` method" and print out the value returned by the method. You should check that the return values of your implementation of `getCell` agrees with those found in the result column of the table. For example, to test the first pattern in the table you may like to use the following code:

```
System.out.println(getCell(world,1,1));
```

world	x	y	result
0x20A0600000000000L	1	1	false
0x20A0600000000000L	-1	1	false
0x20A0600000000000L	3	8	false
0x20A0600000000000L	1	-12	false
0x20A0600000000000L	72	2	false
0x20A0600000000000L	5	5	true
0x20A0600000000000L	5	6	true
0x20A0600000000000L	6	6	false

**Table 1. Test patterns for the `getCell` method**

Type in the following as a new method into your class `TinyLife`:

```
public static void print(long world) {
    System.out.println("-");
    for (int y = 0; y < 8; y++) {
        for (int x = 0; x < 8; x++) {
            System.out.print(getCell(world, x, y) ? "#" : "_");
        }
        System.out.println();
    }
}
```

Do not worry about the use of `for` statement at the moment; you will find out what the `for` loop does in the next section. The `print` method allows you to print out a textual "picture" of the state of the world to the terminal. You should test this function by using it to print out the two example worlds as shown in Figure 3, "Example `TinyLife` worlds". Try this now by calling the `print` function from within the `main` method of `TinyLife` and providing a suitable value for the parameter `world`.

In order to step forward in time, we need a way of updating the cells of the world. Add a method called `setCell` to your class `TinyLife` with the following prototype:

```
public static long setCell(long world, int x, int y, boolean value)
```

The `setCell` method should use the `PackedLong.set` method to update the cell  $(x,y)$  stored in `world` with the value `value`. The method should return the updated version of `world` at the end of the method body. Please ensure that your implementation of `setCell` performs input sanitisation where necessary.

You should make use of the `print` method to make sure that your `setCell` method works correctly. (Hint: initialise a variable called `world` with a well-known pattern, update the variable with a call to `setCell`, such as `"world = setCell(world, 1, 1, true)"`, and print out the contents of `world` to confirm the correct cell was updated.)

## Updating the world

In this section you will use the two methods you wrote in the previous section to apply the rules of the Game of Life to the world and explore different generations. There are three basic tasks to perform when applying the rules of the Game of Life to calculate generation  $t+1$  from the state of the world in generation  $t$ . These are: (1) for each cell, count the number of neighbours; (2) for each cell, given the number of neighbours, compute whether the cell lives or dies; and (3) generate a new data structure to represent the next generation. We can capture these three phases into three methods:

- `public static int countNeighbours(long world, int x, int y)`
- `public static boolean computeCell(long world,int x, int y)`
- `public static long nextGeneration(long world)`

You will complete implementations of these three methods in this section. Notice that these methods work together to provide a solution to calculating the state of the next generation: `computeCell` should use `countNeighbours`, and `nextGeneration` should use `computeCell`. Your first task in this section is to complete the method body for the function `countNeighbours`. The method receives a description of the game board as the variable `world` together with a specific indexed cell  $(x,y)$ . Your implementation should count the number of neighbours which are alive in the eight locations immediately adjacent to the cell  $(x,y)$ . For example, if your method is called as follows:

```
countNeighbours(0x20A0600000000000L,6,6);
```

then your program should return the number of live cells which are adjacent to cell  $(6,6)$ , which in this case is 5. (You can check that is 5 the correct answer by viewing world `0x20A0600000000000L` in Figure 3, "Example `TinyLife` worlds".)

6. Create a table similar to Table 1, "Test patterns for the `getCell` method" with five test cases for the method `countNeighbours`. Your table will need four columns: `world`, `x`, `y` and *result*. You should use the world `0x20A0600000000000L` in all your test cases and make sure you select cells which have 0, 1, 2 and 5 neighbours. You will probably find it helpful to refer to Figure 3, "Example `TinyLife` worlds".

7. Use the table you have just created to test your implementation of `countNeighbours`.

Your next task in this section is to complete the implementation for `computeCell` which should use your implementations of `getCell` and `countNeighbours` to compute the value that the specified

cell should have in the next generation. A skeleton implementation of the method is shown below. You should complete the sections marked `TODO`.

```
public static boolean computeCell(long world,int x,int y) {

    // liveCell is true if the cell at position (x,y) in world is live
    boolean liveCell = getCell(world, x, y);

    // neighbours is the number of live neighbours to cell (x,y)
    int neighbours = countNeighbours(world, x, y);

    // we will return this value at the end of the method to indicate whether
    // cell (x,y) should be live in the next generation
    boolean nextCell = false;

    //A live cell with less than two neighbours dies (underpopulation)
    if (neighbours < 2) {
        nextCell = false;
    }

    //A live cell with two or three neighbours lives (a balanced population)
    //TODO: write a if statement to check neighbours and update nextCell

    //A live cell with with more than three neighbours dies (overcrowding)
    //TODO: write a if statement to check neighbours and update nextCell

    //A dead cell with three live neighbours comes alive
    //TODO: write a if statement to check neighbours and update nextCell

    return nextCell;
}
```

8. Create a table similar to Table 1, "Test patterns for the `getCell` method" which contains at least eight test cases for the method `computeCell`. Your table will need four columns: `world`, `x`, `y` and *result*. You will probably find it helpful to refer to Figure 3, "Example TinyLife worlds" when defining your test cases.

9. Use the table you have just created to test your implementation of `computeCell`.

## Looping constructs

The `print` method you saw earlier uses a new construct you haven't yet seen in Java: a `for` loop. In general the `for` loop has the following syntax:

```
for (initialisation; boolean_expression; step)
    statement
```

The `initialisation` section is used to initialise a new or existing variable (if any). The `boolean_expression` is evaluated to determine if the body of the `for` loop should be executed (`true` implies the body is executed and `false` implies otherwise); `boolean_expression` is re-evaluated every time the loop is executed. Finally, `step` is an expression which is executed *after* every iteration through `statement`. As you've seen for the `if` statement already, you can use curly brackets (`{` and `}`) to group together multiple Java statements together into a basic block, and use the basic block in place of `statement`.

Another useful construct in Java for executing the same Java statement repeatedly is a `while` loop. The `while` loop has a simpler syntax than the `for` loop:

```
while (boolean_expression)
    statement
```

The `while` loop will repeatedly execute `statement` while `boolean_expression` evaluates to `true`; if `boolean_expression` evaluates to `false` then execution continues with the next Java statement immediately following `statement`. You can use a basic block to place multiple Java statements inside the body of the `while` loop.

Write an implementation of the `nextGeneration` method. The method accepts a value of type `long` which represents the state of the world at generation `t`; the method should return a value of type `long` which represents the state of the world at generation `t+1`. The `nextGeneration` method should make use of the `computeCell` method to calculate the future state of a particular cell, and then use the `setCell` method to update a local copy of the world held in a variable of type `long`. You need to perform this action on *all* 64 cells in `TinyLife`. Rather than writing code to update all 64 cells individually, you should use a pair of nested `for` loops. (Hint: start off with a copy of the `print` method as the body for `nextGeneration` and rather than printing out the contents of each cell, use the `for` loops to update each cell stored in a local variable representing the world at generation `t+1`.)

## Displaying a Game of Life

Add the following method to `TinyLife`:

```
public static void play(long world) throws Exception {
    int userResponse = 0;
    while (userResponse != 'q') {
        print(world);
        userResponse = System.in.read();
        world = nextGeneration(world);
    }
}
```

and update your copy of the `main` method to read:

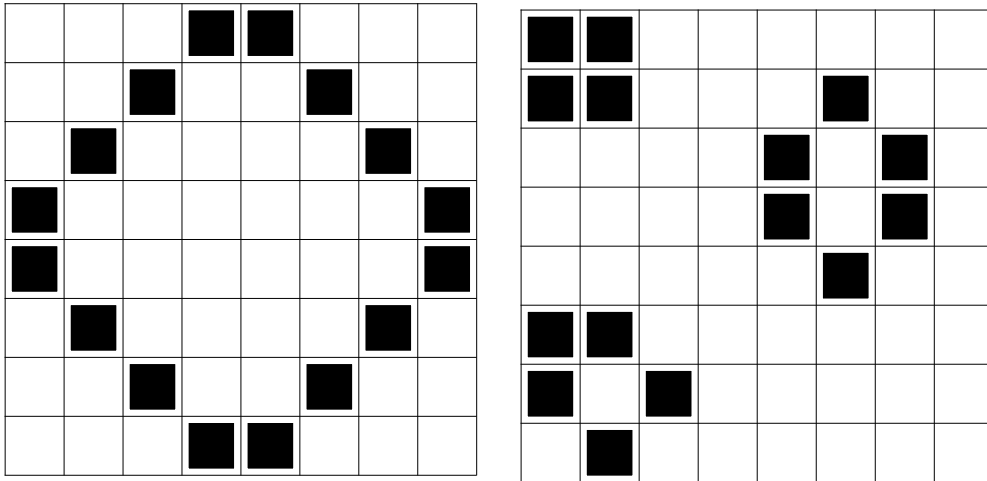
```
public static void main(String[] args) throws Exception {
    play(Long.decode(args[0]));
}
```

If you have written your implementations of the five functions `getCell`, `countNeighbours`, `computeCell`, `setCell` and `nextGeneration` correctly and included the provided methods `print`, `play` and `main` you should now be able to invoke your Java program from the command line and supply the a description of the initial state of the work as a `long` literal value. For example:

```
crsid@machine:~> java uk.ac.cam.crsid.TinyLife 0x20A0600000000000
```

should display a world containing a *glider*. Pressing **Enter** will display successive generations of the world, and typing **q** followed by **Enter** will terminate your program. Note that when running your program from the command line you should not type the trailing `L` on the end of a `long` literal.

Figure 4, “Example `TinyLife` worlds” contains two further examples which you may wish to try. The on-line version of this work book has animated versions of these worlds which you may wish to view in order to confirm your program is operating correctly.



Left, a *five phase oscillator* (code: 0x1824428181422418L)  
 and right, a *still life* (code: 0x205032050502303L).

**Figure 4. Example TinyLife worlds**

## Java Tick 3

In the course of this Tick you should have written five methods: `getCell`, `countNeighbours`, `computeCell`, `setCell` and `nextGeneration`, as well as including three provided methods `print`, `play` and `main` into a class called `TinyLife`. In addition you should also have produced an `answers.txt` file and copied across your implementation of `PackedLong` from last week.

To submit your tick for this week, produce a jar file called `crsid-tick3.jar` with the following contents:

```
META-INF/  
META-INF/MANIFEST.MF  
uk/ac/cam/crsid/tick3/TinyLife.class  
uk/ac/cam/crsid/tick3/TinyLife.java  
uk/ac/cam/crsid/tick3/PackedLong.class  
uk/ac/cam/crsid/tick3/PackedLong.java  
uk/ac/cam/crsid/tick3/answers.txt
```

The jar file should have its entry point set to `uk.ac.cam.crsid.tick3.TinyLife` so that you can invoke `TinyLife` from the command line as follows:

```
crsid@machine:~> java -jar crsid-tick3.jar 0x20A0600000000000
```

Once you have produced a suitable jar file and tested that it works as shown above, you can submit it by emailing it as an attachment to `ticks1a-java@cl.cam.ac.uk`.

