Workbook 3

Introduction

Last week you built a working implementation of the Game of Life using your implementation of `PackedLong` to store the state of the world in a variable of type `long`. The use of `PackedLong` limits the size of world you can store to an eight-by-eight grid of cells. This week you will learn about Java arrays which can be used to store arbitrary-sized blocks of data. You will use Java arrays to write a new version of the Game of Life which can store a world of an arbitary size. As the size of the world gets bigger, initialising the world using a long value becomes inappropriate, so you will design your own object to store the initial starting pattern of the world and decode it from a simple string representation. By the end of this Workbook you should be familiar with creating and using arrays and objects, and the use of strings in Java.

Important

The recommended text book for this course is *Thinking in Java* by Bruce Eckel. You can download a copy of the 3rd Edition for free from Bruce's website:

http://www.mindview.net/Books/TIJ/

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

http://www.cl.cam.ac.uk/teaching/current/ProgJava

You will find the Java standard library documentation useful:

http://java.sun.com/javase/6/docs/api/

Arrays

An array in Java is used to store a fixed number of elements of the same type. New arrays can be created either by using the Java `new` operator or by using curly brackets ({ and }) to provide an explicit list of initial values for the array. Here are two examples:

```
//create a new array with five elements, each initialised to zero
int[] intArray = new int[5];

//create a new array and initialise the elements with the provided values
long[] longArray = {1L, 2L};
```

Note the use of the square brackets after the type to denote the fact that this is an array of values of the specified type. For example, `int[]` tells the compiler that the variable `intArray` is of type "array of ints" rather than simply "int". You can only use the curly bracket notation when creating a *new* variable of an array type; you cannot use it to update the values stored in an existing variable. For example:

```
//This is okay; create new array with 4 elements
long[] longArray = {1L, 2L, 3L, 4L};

//This is wrong; you cannot update values in an array this way
longArray = {1L, 2L, 3L, 4L};
```

You might like to try this to see the error from the compiler. Square brackets are used to retrieve or update values stored in the array. By convention, the first element in the array has an index value of zero, not one. Here are some examples:

```
//create a new array with five elements, each initialised to zero
int[] intArray = new int[5];

//set the value of the FIRST element to 2
intArray[0] = 2;

//set the value of the THIRD element to 9
intArray[2] = intArray[0] + 7;

//increment the value stored in the FOURTH element to 1
intArray[3]++;
```

The square bracket notation only permits a single array value to be updated at a time. Consequently it is a common to use a `for` loop to update values stored in an array. Here is an example:

```
int[] numbers = new int[5];
for(int i=0; i<numbers.length; i++)
 numbers[i] = i;
```

In the above example the `for` loop is used to update each value of the "array of ints" `numbers` in turn with the current value stored in the variable `i`. Notice the use of the phrase `numbers.length`; the value stored in the `length` field is a *read-only* value of the size of the array. It is a good idea to use the `length` field when using Java arrays, rather than using literal values. For example, `numbers.length` should be used rather than the literal `5` because `numbers.length` will always return the length of the array correctly, whereas `5` might be wrong if you subsequently modify the size of the `numbers` array later.

By creating a suitable directory structure and file of the correct name, create a class called `FibonacciCache` inside the package `uk.ac.cam.crsid.tick3`. Use the following code as the basis for writing the class `FibonacciCache`:

```
package uk.ac.cam.crsid.tick3;

public class FibonacciCache {
 public static long[] fib = new long[20];

 public static void store(int n) {
  //TODO: using a for loop, update the value of fib
  //      with the first n values of the Fibonacci sequence
  //      e.g. store(0) should store nothing; and
  //           store(6) should store 1,1,2,3,5,8 in first six elements of fib
  //Note: your code should check that n will fit inside fib
  //      and if it is outside this range perform no action
 }

 public static void reset() {
  //TODO: using a for loop, set all the elements of fib to zero
 }

 public static long get(int i) {
  //TODO: return the value of the element in fib found at index i
  //      e.g. "get(3)" should return the fourth element of fib
  //
  //Note: your code should check that i is within the bounds of fib
  //      and if it is outside this range return the value of fib[0]
 }
}
```

Write an implementation of FibonacciCache

1. Complete the implementation of the three methods in `FibonacciCache` following the instructions inside the comments.

Hint: You will probably want to test your code by writing a "main" method and including some test code which uses your `reset`, `store` and `get` implementations and checks they function correctly. Printing out the contents of `fib` after performing each method call will probably help you find errors.

References

In the Java programs you wrote for Workbooks 1 and 2, the types of the variables you used were all primitive types (i.e. `boolean`, `byte`, `short`, `char`, `int`, `long`, `float` or `double`). In Java, the mechanism used to store values in variables of primitive type differs from the mechanism used to store values other types. Take a look at the following piece of Java code:

```
class Reference {
 public static void update(int i, int[] array) {
  i++;
  array[0]++;
 }
 public static void main(String[] args) {
  int test_i = 1;
  int[] test_array = {1};

  update(test_i, test_array);

  System.out.println(test_i);
  System.out.println(test_array[0]);
 }
}
```

It might surprise you to learn that this program prints out `1` followed by `2` to the terminal. Try it for yourself now.

The reason for the difference is that variables of primitive type store the value directly, where as variables for all other types in Java store a *reference* to the location in memory in which the data is held. In the example above, when the method `update` is called, the variable `i` is initialised with a copy of the value of `test_i`, so all updates to `i` inside the body of the method are lost when the method finishes; in contrast, the variable `array` is initialised with a *reference* to the location in computer memory that `test_array` also references, and this location in computer memory is used to store the contents of the array. Consequently, we say that primitive values in Java are *passed-by-value* and arrays (and as we shall see later, objects) in Java are *passed-by-reference*.

The ability to define references provides the possibility of referencing nothing at all. In Java, you can do this by pointing a reference to `null` which is written as follows:

```
int[] array = null;
```

You may update a reference field or variable in Java to point to `null` at any point. This isn't an erroneous state, and can be useful, however since the reference does not refer to a valid array (or object), then you cannot access array elements (or object methods or fields); a program which attempts to do so will fail with an error when the program runs. For example, the following fails

```
int[] array = {1,2,3};
array = null
System.out.println(array[1]);
```

because `array` does not have a valid reference on the third line when the square brackets are used to find an element in the array.

In Java we can create two-, three- or higher-dimensional arrays. A Java two-dimensional array is simply an array of *references* to one-dimensional arrays. Here are some example multi-dimensional array defintions:

```
//create a 2-by-2 array with all values initialised to zero
int[][] i = new int[2][2];

//2D array of values with each 1D value of a different length
int [][] j = {i[1],{1,2,3},{4,5,6,7}};

//create a 3D array using two 2D array references
int [][][] k = {i,j};
```

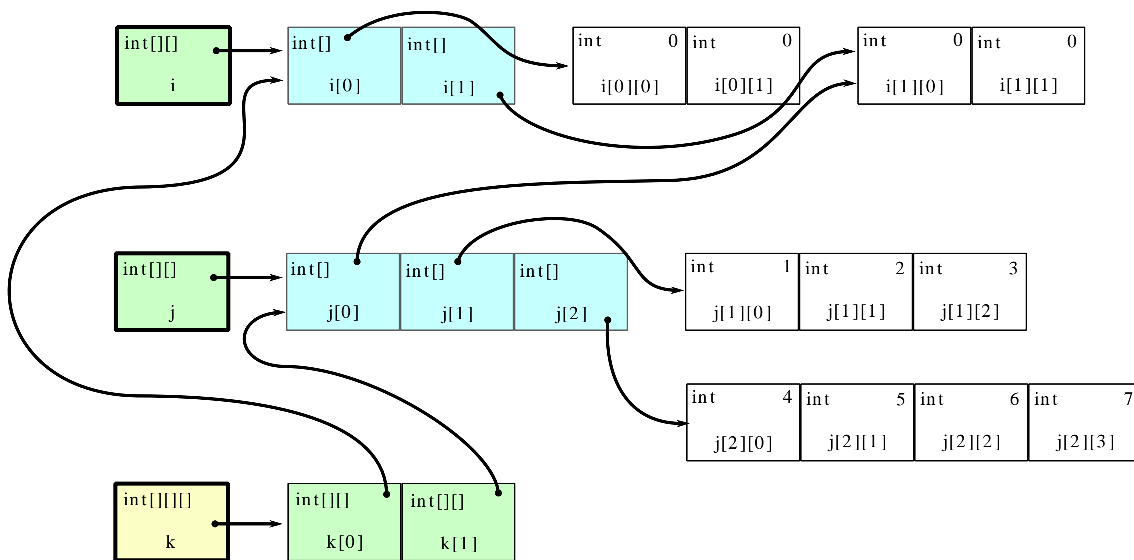

Figure 1. Graphical representation of `i`, `j`, and `k`

The layout of data associated with variables `i`, `j` and `k` is shown in Figure 1, “Graphical representation of `i`, `j`, and `k`”. Notice that some of the data held in the arrays can be referenced using more than one variable. For example `k[0][1][0]`, refers to the same piece of memory as `j[0][0]`, `i[1][0]` and even `k[1][0][0]`! Consequently, updating the value stored using any reference will ensure the updated value is available through all other references. If a piece of data can be accessed through more than one reference, we say that the references are *aliased*.

Write an implementation of ReferenceTest

By creating a suitable directory structure and file of the correct name, create a class called `ReferenceTest` inside the package `uk.ac.cam.crsid.tick3`. Enter the following code as the basis for the class `ReferenceTest`:

```
package uk.ac.cam.crsid.tick3;

public class ReferenceTest {
 public static void main(String[] args) {
  //TODO: Insert the definitions of i,j and k as shown above here

  System.out.println(k[0][1][0]++);
  System.out.println(++k[1][0][0]);
  System.out.println(i[1][0]);
  System.out.println(--j[0][0]);
 }
}
```

2. Copy the definitions of `i`, `j`, and `k` as defined earlier in this section of the workbook into the top of the `main` method.

3. What do you think the program will print when it is executed? Execute the program to check. Your Ticker may ask you what the output of the program is next week.

ArrayLife

Your next task is to write a new implementation of Conway's Game of Life, this time using a two-dimensional array of `boolean`s to store the state of each cell, rather than using your implementation of `PackedLong`. Using a two-dimensional array will allow you to simulate worlds which are larger than an eight-by-eight grid of cells. You should store your data in row order in the two-dimensional array, which means the first array index refers to height (`y`) and the second index refers to width (`x`); see the new version of `getCell` below for a concrete example.

Copy the code you wrote last week in `TinyLife.java` into a file called `ArrayLife.java`, update the name of the class to `ArrayLife`, and place the file inside the package `uk.ac.cam.crsid.tick3`. You should also copy across your implementation of `PackedLong` into the same package. To finish updating your implementation, change each of the method prototypes in your file to the following:

- `public static boolean getCell(boolean[][] world, int x, int y)`
- `public static void setCell(boolean[][] world, int x, int y, boolean value)`
- `public static void print(boolean[][] world)`
- `public static int countNeighbours(boolean[][] world, int x, int y)`
- `public static boolean computeCell(boolean[][] world, int x, int y)`
- `public static boolean[][] nextGeneration(boolean[][] world)`
- `public static void play(boolean[][] world) throws Exception`

Notice that `setCell` does not return an updated version of `world`. This is because arrays are passed by reference, so your implementation for this method can simply update the contents of `world`, safe in the knowledge that these changes will be available when the method returns. *If this concept troubles or puzzles you, please ask the Lecturer or Demonstrator to explain this to you—it's crucial that you understand this concept so don't feel afraid to ask for help!* You will now need to replace all calls to `PackedLong.set` and `PackedLong.get` with suitable operations on a two-dimensional array of booleans. For example, the `getCell` method could be written as:

```
public static boolean getCell(boolean[][] world, int x, int y) {
 if (y < 0 || y > world.length - 1) return false;
 if (x < 0 || x > world[y].length - 1) return false;

 return world[y][x];
}
```

Make sure you understand the difference between `world.length` and `world[y].length`. You must update the body of *all* the methods in ArrayLife as listed above. Finally, replace your `main` method with the following code:

```
public static void main(String[] args) throws Exception {
 int size = Integer.parseInt(args[0]);
 long initial = Long.decode(args[1]);
 boolean[][] world = new boolean[size][size];
 //place the long representation of the game board in the centre of "world"
 for(int i = 0; i < 8; i++) {
  for(int j = 0; j < 8; j++) {
   world[i+size/2-4][j+size/2-4] = PackedLong.get(initial,i*8+j);
  }
 }
 play(world);
}
```

You should then be able to try the game boards listed in Workbook 2 as follows:

```
crsid@machine:~> java uk.ac.cam.crsid.tick3.ArrayLife 12 0x1824428181422418
```

In this case, you should see the five-phase oscillator in the centre of a 12-by-12 world. If you are not doing so already, you should make sure your code is presented in a neat and consistent style so that the structure of the different blocks (delimited by { and }) is clear. Please ask if you're having trouble with presentation.

Write an implementation of ArrayLife

4. Write an implementation of ArrayLife which, given an identical world and an 8-by-8 grid of cells, produces the same output as your implementation of TinyLife. Check this is the case by trying several game boards found in Workbook 2.

Note: this kind of editing is called *refactoring* and is a common technique used by programmers. When you refactor something, you take a piece of code that works, improve or generalise the structure of the code in some way, and then test that it produces the same output as before.

Review of object-oriented programming

The Object-Oriented Programming course covers the essential concepts in object-oriented programming and provided some examples in Java. This section briefly reviews these concepts to give you the skills to improve your implementation of ArrayLife further.

An object is used to encapsulate both a data structure and a set of operations which can be performed on it. Data is stored in *fields*, and operations are written as *methods*. For example, we might like to create an object to represent a cell in the Game of Life. A cell object might have a field to record whether the cell is alive or dead; the same object can also provide methods for changing the state of the cell to "alive" or "dead".

An object-oriented programming language needs some method of describing an object, and in Java this is done by writing a *class*. To create objects which represent cells in the Game of Life, we would define

a class to encapsulate the data (the current state of the cell, alive or dead) and the actions we can perform on it (changing the state of the cell to alive or dead); from a practical perspective we probably need a method of retrieving the current state of the cell too! Once we have a class definition we can create multiple instances of a class, each of which will have an independent data store.

One possible definition of a Cell object in Java is as follows:

```
class Cell {
 boolean alive;
 Cell() {alive=false;}
 boolean isAlive() {return alive;}
 void create() {alive=true;}
 void kill() {alive=false;}
}
```

where each of the lines in the above code perform the following actions:

- The first line declares a new class called `Cell`; the declaration includes everything between lines 2 and 6 and is terminated with the closing curly bracket (`}`) on line 7.
- The second line declares the desire for space in computer memory to store information which has the name `alive` and has the type `boolean`.
- The third line is a constructor for the class (a method which is executed when an object of the class is created); in this case the constructor initialises the field `alive` to `false`.
- The fourth line defines a method which returns the current state of the cell.
- The fifth line defines a method which updates the state of the cell to alive.
- The sixth line defines a method which updates the state of the cell to dead.

The definition of the class `Cell` has not actually created any cell objects yet, it merely provides a schematic or plan of how to create them; we have created a new *type*. We can use a class definition to create multiple instances of a class which, as we've already seen, are often called objects.

To make use of a class definition we need to create an instance of it (those familiar with Java might complain that there are some exceptions to this rule—`static` fields and methods— but we'll ignore these for now). We also need some method of naming or referring to the specific instance, which is called a *reference*. Creating an instance of a class, and keeping a reference to it are distinct concepts in Java since it is often useful to have several references pointing to a single instance. In fact, you've already used references in Java earlier in this workbook in the context of Java arrays. If an instance of an object exists which has no reference to it, the programmer has no way of referring to it and it is therefore no longer useful; in this case the Java runtime can safely delete the instance. This process is called *garbage collection* and is handled automatically in Java.

We can create a new instance of an object by calling the class constructor, prefixed with the `new` key-word; for example "`new Cell()`". If our class definition does not contain any constructors, the Java compiler will create a default constructor for us; a default constructor takes no arguments. We can create space to store a reference which points to an object by writing the type (the name of the class) followed by a memorable name which we can use to refer to the reference, for example, "`Cell cell1`"; this storage space is called a *field* if it appears inside the body of a class definition, or a *variable* if it appears inside the body of a method. A variable or field can reference an object in Java by using the assignment operator (=), and we can do all these steps in one line, or separately:

```
Cell cell1 = new Cell();
Cell cell2;
cell2 = new Cell();
```

In the above example,

- the first line creates a variable called `cell1` of type `Cell` and updates the contents of the variable to reference a new instance of the `Cell` class.
- The second line simply creates a new variable called `cell2` which at this stage does not point to any instance;
- the third line updates the variable `cell2` so that it references a second instance of `Cell`.

As you will have seen in lectures, we can use variables or fields which reference objects to invoke methods and access fields on those objects.

Static fields and methods

Usually fields and methods are associated with instances of a class. In other words, each object has it's own copy of a field and method. Sometimes however you may want *every* object or instance of a particular class to share the *same* field or method. In this case you prefix the name of the field or method with the keyword `static`.

Up until now you have only been using static methods. For example, the special `main` method is declared `static` because the Java runtime invokes it without creating an instance of the class which contains it. Because a static method or field is not associated with any particular instance, you refer to it by using the name of the class *not* the name of the instance. For example, last week you wrote `PackedLong.get` and `PackedLong.set` to refer to static methods associated with the class `PackedLong`.

Passing method arguments

Just as we have seen for Java arrays, arguments and return types of objects are passed using references to the same underlying object instance; therefore using an object instance as an argument to a method does *not* make a copy of the object. As mentioned earlier in the second section of this Workbook, this is called *call-by-reference*. Many procedural and object-oriented programming languages support this type of argument passing.

If you do not wish to return a value from a method, then specify a return type of `void`. For example, the special main function you've seen in Java has a return type of `void`. This is a bit like `unit` in ML.

Naming conventions and reserved words

You should now be able to write simple classes in Java which contain fields, methods and variables. You should choose names for such things carefully, partly because a good name for a class, method or variable will improve readability (and therefore is strongly encouraged from a software engineering perspective), but also because some phrases cannot be used for classes, methods, fields or variable names in Java. You may recall from Workbook 2 that the following names are *reserved* words in Java:

```
abstract   assert       boolean*   break     byte*      case
catch      char*        class*     const     continue   default
do         double*      else*      enum      extends    false*
final      finally      float*     for*      goto       if*
implements import       instanceof int*      interface  long*
native     new*         null*      package*  private    protected
public     return*      short*     static*   strictfp   super
switch     synchronized this       throw     throws     transient
true*      try          void*      volatile  while*
```

The keywords marked with an asterisk (`*`) are terms which you should recognise.

Using any of these words as a name for a class, method, field or variable name will result in a compile error. If you encounter this problem, the solution is to change the name so it is no longer a reserved

word. You will learn what many of the other reserved words do in this course, but not all. Some of the advanced features of Java will not be taught until the second year.

Every class object has some built-in or special method names which you should also avoid using. These are:

```
clone equals finalize getClass notify notifyAll toString wait
```

You should avoid naming a method with one of these names unless you really know what you're doing. Unfortunately you may not get a compile error if you use one of these names for a method accidentally. This is because there are circumstances when you may wish to provide your own implementation of these methods. Therefore if your program appears to behave incorrectly, you should double-check to make sure none of your methods have one of these names.

By convention, class names in Java should start with an upper case letter, and variable, field and method names should begin with a lower case letter. Consistent use of capitalisation will help you differentiate between class names and variable names, and may also aid syntax highlighting in some editors.

This section reviewed important concepts and terminology. You should make sure you know what the following terms mean before continuing: class, instance, object, field, method, static, variable, constructor, garbage collection, pass-by-reference, pass-by-value, `new`.

Strings

Your implementation of ArrayLife is still limited to simulating a game board with an eight-by-eight grid of cells because it uses values of type `long` to initialise the world. To explore larger and more interesting worlds, you will need to use bigger patterns. To describe larger worlds you will use an instance of the standard library `String` class. You can see the methods available for objects of type `String` by looking them up in Sun's documentation. Do this now by visiting the following website

http://java.sun.com/javase/6/docs/api/

and find the `String` class in the list of "All Classes" in the lower left-hand pane. You will need to refer to this documentation later in this tick. The `String` class is used extensively in the Java library, and consequently has two special features of importance in this course. These are

- an instance of `String` can be created without using the `new` operator and a call to the constructor: simply place the string pattern inside double quotes, for example `"Alastair Beresford";`
- the addition operator, when combined with a string, can be used to convert and append variables or literals of another type to a string, for example `"The University of Cambridge is "+800+" years old"` creates a new string whose contents is `The University of Cambridge is 800 years old`.

A Java `String` is immutable, so the contents cannot be updated after initialisation; instead, all methods associated with String object create a new instance of the `String` class which contains the relevant updates. Here are some example ways of creating `String` objects and variables which reference them:

```
String s1 = new String("Cambridge");
String s2 = s1; //s2 and s1 now reference the same object instance
String s3 = "University of "; //new instance of String, but no "new" operator!
String s4 = s3 + s2; //concatenate two strings
String upper = s4.toUpperCase(); //Note: contents of s4 is distinct from upper
String lower = "University of Cambridge".toLowerCase();
```

In the remainder of this course you will use a Java `String` to describe the world in Conway's Game of Life. There are a multitude of sensible patterns you could use to describe the game board, but you should use the format described here so that your program is compatible with the examples in this Workbook and the automated tests. The format you need to support is

```
NAME:AUTHOR:WIDTH:HEIGHT:STARTX:STARTY:CELLS
```

where NAME is the name given to the board layout, AUTHOR is the crsid or name of the author, and WIDTH and HEIGHT describe the board dimensions. Rather than specifying the state of all cells in the world, the format assumes that cells are dead unless explicitly specified otherwise. Consequently CELLS is used to represent a *subset* of the board which contains all the live cells. (Most cells in a typical world are dead so this optimisation is frequently useful.) The values of STARTX and STARTY specify the location of the cells recorded in CELLS in the world. The details of the format are perhaps best explained with the aid of the following example:

```
Glider:Richard Guy (1970):20:20:1:1:010 001 111
```

The above example describes a world of 20 cells by 20 cells with a "Glider" in it. Gliders were discovered by Richard Guy in 1970. The contents of CELLS is 010 001 111 and records live cells with a one (1) and dead cells with a zero (0) in row order, using spaces to separate rows. Therefore, in the above example, the CELLS part of the format states that cells (1,0), (2,1), (0,2), (1,2) and (2,2) are alive. The values of STARTX and STARTY should be added to the values recorded in CELLS and therefore the live cells in the world at generation zero are (2,1), (3,2), (1,3), (2,3) and (3,3); all other cells are dead.

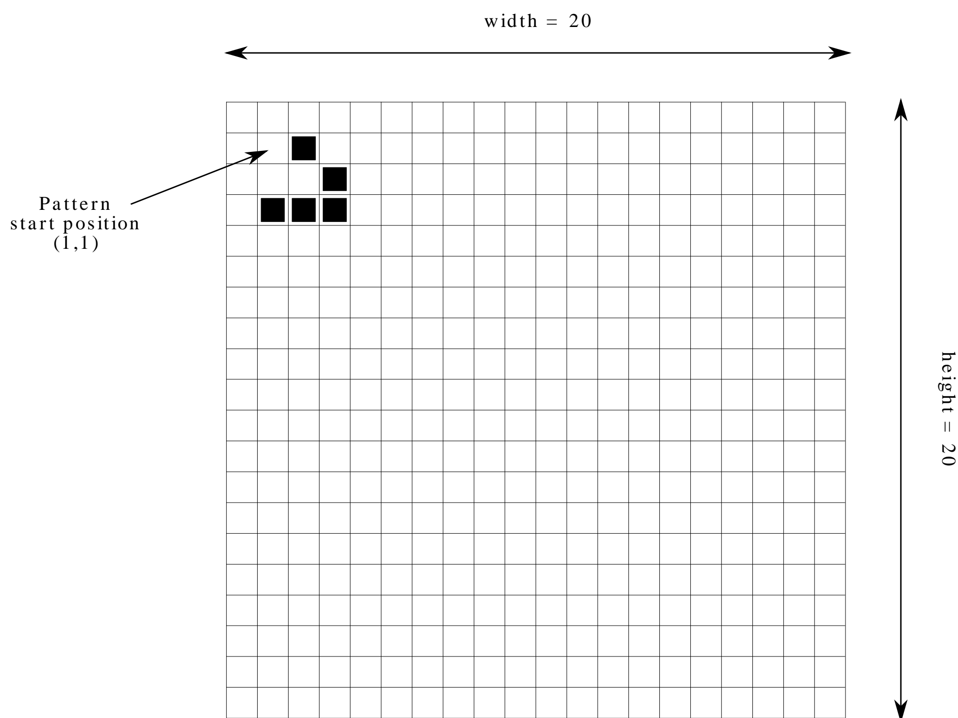

Figure 2. The Glider in a 20-by-20 world

The layout of this world is shown in Figure 2, “The Glider in a 20-by-20 world”. Remember that this format assumes that cells are initialised as *dead* unless they are explicitly marked as alive in the CELLS section of the format string.

Your next challenge is to update your program ArrayLife so that it accepts strings in the new format. This updated version should be called StringArrayLife. StringArrayLife should accept the appropriately formatted string on the command line and use it to initialise the state of the world at generation zero. For example,

```
crsid@machine:~> java uk.ac.cam.crsid.tick3.StringArrayLife \
"Glider:Richard Guy (1970):20:20:1:1:010 001 111"
```

It's essential to use the double quotes (") around the format string, otherwise the shell will interpret the format string as multiple arguments rather than as a single argument. In order to support the string format, you will need to devise a method of parsing and interpreting the values in the format string. Below are some code snippets which you may find helpful. You should use Sun's on-line documentation to work out how to use them.

- `"A:B:C".split(":")`
- `"001".toCharArray()`
- `"010 001 111".split(" ")`
- `Integer.parseInt("1")`

You may like to check your understanding of the code snippets by writing a simple test program.

Creating StringArrayLife

5. Copy the contents of your implementation of `ArrayLife.java` into a new file called `StringArrayLife.java` and update the name of the class accordingly.

6. Replace the contents of the `main` function with the following code snippet and complete the sections marked `TODO`.

```
public static void main(String[] args) throws Exception {

 String formatString = args[0];

 //TODO: Determine the dimensions of the game board
 int width = ...
 int height = ...
 boolean[][] world = new boolean[height][width];

 //TODO: Using loops, update the appropriate cells of "world"
 //      to "true"
 // ...

 play(world);
}
```

7. Check your implementation of StringArrayLife works by testing it with the format string: `"Glider:Richard Guy (1970):20:20:1:1:010 001 111"`. You should ensure that the initial state of the game is as shown in Figure 2, “The Glider in a 20-by-20 world”.

Writing your own Object

In the final part of this workbook, you are going to redesign the StringArrayLife class to reduce the tight-coupling which currently exists in its design. Assume for a moment that you were working on your simulation of Conway's Game of Life with another developer. One developer might be working on parts of the program which applies the game rules to the two-dimensional world array, and the other developer on the subsystem used to load patterns and initialise the world. With the current design both developers will need to inform each other whenever they make almost any change.

Good abstraction is essential in software development and in an object-oriented programming language objects provide the primary method of abstraction. In Java, a well designed class should allow a programmer to interact with it as a black box without worrying about how it works—this is beneficial for multiple people working on a project and also beneficial to a single programmer because it means they can write clearer programs which are more likely to function correctly. Therefore as the last task in this Workbook you will create a new class called `Pattern` which will encapsulate all the state expressed by the format string described in the last section.

As you will see shortly, in this class you will prefix field names with the keyword `private`. This means that the fields are only accessible by methods associated with the class, and not from elsewhere. This is a form of *encapsulation* since it prevents external, uncontrolled modification of internal state. It is common to write "get" methods associated with some private fields which provide read-only access to the underlying state.

By creating a suitable directory structure and file of the correct name, create a class called `Pattern` inside the package `uk.ac.cam.crsid.tick3`. Use the following code as the basis for writing the class `Pattern`:

```
package uk.ac.cam.crsid.tick3;

public class Pattern {

 private String name;
 private String author;
 private int width;
 private int height;
 private int startX;
 private int startY;
 private String cells;
 //TODO: write public "get" methods for EACH of the fields above;
 //      for instance "getName" should be written as:
 public String getName() {
  return name;
 }

 public Pattern(String format) {
  //TODO: initialise all fields of this class using contents of "format"
  //      to determine the correct values.
 }

 public void initialise(boolean[][] world) {
  //TODO: update the values in the 2D array representing the state of "world"
  //      as expressed by the contents of the field "cells".
 }
}
```

The behaviour of `Pattern` when given an incorrectly formatted string is undefined for the purposes of this Tick, however you may like to choose some sensible defaults. The automatic tests associated with this Tick will always provide well formatted strings.

By creating a suitable directory structure and file of the correct name, create a class called `PatternLife` inside the package `uk.ac.cam.crsid.tick3`. Copy the contents of `StringArrayLife.java` into `PatternLife.java` and replace the `main` function with:

```
public static void main(String[] args) throws Exception {

 Pattern p = new Pattern(args[0]);
 boolean[][] world = new boolean[p.getHeight()][p.getWidth()];
 p.initialise(world);
 play(world);
}
```

8. Complete your implementation of `Pattern`.
9. Complete your implmentation of `PatternLife`.
10. Test your implementation of `PatternLife` by providing appropriate configuration strings, for example:

```
crsid@machine:~> java uk.ac.cam.crsid.tick3.PatternLife \
"Glider:Richard Guy (1970):20:20:1:1:010 001 111"
```

Java Tick 3

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/FibonacciCache.class
uk/ac/cam/crsid/tick3/FibonacciCache.java
uk/ac/cam/crsid/tick3/ReferenceTest.java
uk/ac/cam/crsid/tick3/ReferenceTest.class
uk/ac/cam/crsid/tick3/ArrayLife.java
uk/ac/cam/crsid/tick3/ArrayLife.class
uk/ac/cam/crsid/tick3/StringArrayLife.java
uk/ac/cam/crsid/tick3/StringArrayLife.class
uk/ac/cam/crsid/tick3/Pattern.java
uk/ac/cam/crsid/tick3/Pattern.class
uk/ac/cam/crsid/tick3/PatternLife.java
uk/ac/cam/crsid/tick3/PatternLife.class
uk/ac/cam/crsid/tick3/PackedLong.java
uk/ac/cam/crsid/tick3/PackedLong.class
```

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

```
crsid@machine:~> java -jar crsid-tick3.jar \
"Glider:Richard Guy (1970):20:20:1:1:010 001 111"
```

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