Workbook 6

Introduction

Last week you improved the error handling of your program by catching errors and throwing exceptions. You did this by defining your own exception, `PatternFormatException`, which extended the class `Exception`. This week you will explore what "extending" a class actually means by looking at inheritance in more detail. You will also use this knowledge to refactor your implementation of LoaderLife you wrote last week to separate out the data structure used to record the state of the world into a separate object.

Important

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

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

Interfaces

An *interface* in Java defines a type, and specifies the methods which are *required* in any implementation of the interface. For example, we could state the following methods are required in an implementation which is able to store, update and draw the state of a world in the Game of Life:

```
package uk.ac.cam.acr31.life;

public interface World {
 public void setCell(int x, int y, boolean alive);
 public boolean getCell(int x, int y);
 public int getWidth();
 public int getHeight();
 public int getGeneration();
 public int getPopulation();
 public void print(Writer w);
 public void draw(Graphics g,int width, int height);
 public World nextGeneration(int log2StepSize);
}
```

Particular bits to note in this definition of the interface `World` are the use of the keyword `interface` in the definition (not `class`), and the use of the semicolon (`;`) instead of curly brackets (`{` and `}`) after the method names, indicating that the methods themselves are merely prototypes or descriptions and cannot contain actual code.

The specification of the interface `World` states that any implementation of the interface is required to provide nine methods: `setCell`, `getCell`, `getWidth`, `getHeight`, and so on. Since the interface does not provide an implementation of the methods, the following is wrong and will result in a compile error:

```
World w = new World(); //Wrong; will result in compile error
```

In order to implement an interface in Java you write a class which contains implementations of every method defined in the interface. In addition, you must use the keyword `implements` to name the interface explicitly in the class definition. For example:

```
package uk.ac.cam.crsid.tick6;

import uk.ac.cam.acr31.life.World;
import java.io.Writer;
import java.awt.Graphics;
import java.io.PrintWriter;

public class TestArrayWorld implements World {

 private int generation;
 private int width;
 private int height;
 private boolean[][] cells;

 public TestArrayWorld(int w, int h) {
   width = w;
   height = h;
   // TODO: set generation equal to zero
   // TODO: set cells to reference a new rectangular two-dimensional
   //       boolean array of size height by width
 }

 protected TestArrayWorld(TestArrayWorld prev) {
   width = prev.width;
   height = prev.height;
   // TODO: set generation equal to prev.generation+1
   // TODO: set cells to reference a new rectangular two-dimensional
   //       boolean array of size height by width
 }

 public boolean getCell(int x, int y) { /* TODO */ }
 public void setCell(int x, int y, boolean alive) { /*TODO*/ }
 public int getWidth()  { /*TODO*/ }
 public int getHeight()  { /*TODO*/ }
 public int getGeneration()  { /*TODO*/ }
 public int getPopulation()  { return 0; }
 public void print(Writer w)  { /*TODO*/ }
 public void draw(Graphics g, int width, int height)  { /*Leave empty*/ }

 private TestArrayWorld nextGeneration() {
   //Construct a new TestArrayWorld object to hold the next generation:
   TestArrayWorld world = new TestArrayWorld(this);
   //TODO: Use for loops with "setCell" and "computeCell" to populate "world"
   return world;
 }

 public World nextGeneration(int log2StepSize)  {
   TestArrayWorld world = this;
   //TODO: repeat the statement in curly brackets 2^log2StepSize times
   {
    world = world.nextGeneration();
   }
   return world;
 }

 //TODO: Add any other private methods which you find helpful
}
```

You will be instructed to complete this code later in the workbook. For the moment just read through it and the explanation that follows. Some of the keywords used (such as `private` and `protected`) might be unfamiliar to you. These are explained later.

In Java, a class may implement multiple interfaces, in which case a list of comma-separated interfaces should appear after the keyword `implements` in the class definition.

You can create variables of type `World` which reference an instance of a class which implements the interface `World`. For example, if you've implemented a class called `MyWorldImpl` which implements the `World` interface, then the following is correct

```
MyWorldImpl a = new MyWorldImpl();
System.out.println(a.getPopulation());
World b = a;
System.out.println(b.getPopulation());
```

or more simply

```
World b = new MyWorldImpl();
System.out.println(b.getPopulation());
```

Any class which implements the interface `World` can be assigned to a variable of type `World`. This allows easy substitution of multiple different implementations, providing an extensible method of defining new ways of storing the state of the world in Conway's Game of Life. For example, later on in this Workbook, you will adapt your code from Tick 3 which used a `long` integer to store the state of a world in a class called `TestPackedWorld`. You will also adapt code from Tick 4 and Tick 5 which used a Java array to store the state of the world in a class called `TestArrayWorld`. Since both these classes will implement the `World` interface, you can substitute one implementation in place of another when drawing the state of the world to the terminal or (as we shall see later) a graphical display.

The interface `World` is provided for you inside the package `uk.ac.cam.acr31.life`, as shown in the definition written at the start of this Workbook. This interface, along with several other classes in this package, have been written for you. Using the `World` interface defined in this package enables the interoperability described above—it also allows you to plug in implementations by other students if you wish. To introduce the name "`World`" as short-hand for `uk.ac.cam.acr31.life.World`, you need to write `import uk.ac.cam.acr31.life.World;` at the top of your source files. This has been done for you in `TestArrayWorld`.

To use the interface `World` you also need to download and save the jar file containing the interface definition from http://www.cl.cam.ac.uk/teaching/0809/ProgJava/world.jar. To use the code available in this jar file, you will need to tell the java compiler (`javac`) and the java runtime environment (`java`) the location of the jar file. If you don't tell the compiler or the runtime where the jar file is, the tools won't be able to locate the definition of the `World` interface and your own programs will not compile or run.

You can instruct the Java tools to look inside a jar file by providing an additional argument on the command line. For example, if you have saved `world.jar` to your home directory, then you can tell the Java compiler to look at, and possibly use, the contents of `world.jar` when compiling your own implementation of `TestArrayWorld` as follows:

```
crsid@machine:~> javac -cp /home/crsid/world.jar:. \
uk/ac/cam/crsid/tick6/TestArrayWorld.java
```

The option `-cp` stands for *class path*. The class path in Java is, by default, the set of directories and jar files which are searched for class definitions. By default, the standard library and current working directory are on the class path but, as you've seen above, you may have to manually add to the class path if you wish to work with libraries provided by third parties.

If you want to include multiple directories and jar files on the class path, then you can use the colon (`:`) operator to separate them. In the example above, the classes within `world.jar`, as well as the class files located in the current working directory (represented by "`.`") are put on the class path. (Whilst the current working directory is on the class path by default, it is removed when using the `-cp` option, so you will often need to include it again.)

The `-jar` option you have used in previous weeks is actually just a convenient way of adding the jar file to the class path, and then executing the class specified as the entry point. For example, last week you build a jar file called `crsid-tick5.jar` with the entry point set to `uk.ac.cam.crsid.tick5.LoaderLife`. You can run the main method in `LoaderLife` by executing:

```
crsid@machine:~> java -jar crsid-tick5.jar
```

which is the same as writing

```
crsid@machine:~> java -cp crsid-tick5.jar uk.ac.cam.crsid.tick5.LoaderLife
```

An implementation of the World interface

Create a new package called `uk.ac.cam.crsid.tick6` and copy the contents of `TestArrayWorld` as shown in the previous section into a file with a suitable name inside this package. Copy across your implementation of `LoaderLife` from last week, renaming the class `RefactorLife`. Finally, copy across your implementations of `Pattern`, `PatternLoader` and `PatternFormatException`. Remember to update the package statements at the top of your source files appropriately.

Some further hints on how to complete these questions appear in the Workbook immediately after this question box.

1. Rewrite the `initialise` method of `Pattern` so that it takes a single argument of type `World`. Hint: you will need to make use of `setCell` defined in the `World` interface to initialise the pattern correctly.
2. Complete the sections marked `TODO` in `TestArrayWorld` by adapting the code currently in `RefactorLife` as a template. For example, you might find it helpful to look at the method `setCell` in `RefactorLife` when implementing `setCell` in `TestArrayWorld`.
3. Adapt your implementation of the `main` method in `RefactorLife` to use `TestArrayWorld` to store the current state of the world.
4. If you have done the above correctly you will find that all but the `main` and `play` methods in `RefactorLife` are not used, since you should have copied or adapted all the other methods into `TestArrayWorld`. Delete all the unused methods from `RefactorLife`.
5. Test your refactored program still works by using the procedure provided in the section labelled "Testing your program" below.

Stub methods (draw and getPopulation)

When implementing `TestArrayWorld` you do not have to implement the `draw` method, so as the comment suggests, you may leave it blank. Similarly you will see that `getPopulation` method returns 0 rather than the correct answer. These are commonly known as method stubs. The idea is to put something simple in so that the program will compile and unrelated functionality will work. Once the program compiles the programmer will return to the stubs and incrementally replace them with working code.

The print method

The print method should take a single argument of type Writer rather than simply printing using System.out.print and System.out.println. Here is a skeleton which you can use to complete the new version of print:

```
public void print(Writer w) {
 PrintWriter pw = new PrintWriter(w);
 // See the Java documentation for PrintWriter
 //
 // use pw.print("something") to write to the writer
 // use pw.println("something") to write a newline
 //
 // you must always call pw.flush() at the end of this method
 // to force the PrintWriter to write to the terminal (if you
 // do not, then data may be buffered inside PrintWriter).
}
```

You can use the Java OutputStreamWriter class to create an instance of type Writer which outputs information to the terminal. For example, the following snippet of code should print the contents of a variable world of type World to the terminal:

```
Writer w = new OutputStreamWriter(System.out);
world.print(w);
```

Remember to include appropriate import statements at the top of your source file. For example, you will need to write:

```
import java.io.PrintWriter;
import java.io.OutputStreamWriter;
```

to successfully use the PrintWriter and OutputStreamWriter.

The nextGeneration methods

The World interface specifies a nextGeneration method which takes a single argument called log2StepSize. This argument states the number of generations to iterate through before returning the state of the new world. For example, if log2StepSize has the value 0, then, since $2^0 = 1$, executing the method nextGeneration with the value 0 should update the state of the world by one generation. If log2StepSize has the value 2, then since $2^2 = 4$, executing the method nextGeneration with the value 2 should update the state of the world by four generations. You should use a for loop inside the method nextGeneration to update the state of the world the correct number of times. You might find it helpful to recall that `1 << n` is equivalent to 2^n.

A good way to implement the nextGeneration method described above is to first complete the private nextGeneration method (the one which takes no arguments) which should perform a single timestep forward in the Game of Life. Then complete the original nextGeneration method required by the interface by repeatedly calling your private nextGeneration method. You will need to use the methods computeCell and countNeighbours which are discussed next.

Other methods

In previous weeks you wrote several helpful methods inside the body of what is now called RefactorLife. These methods include countNeighbours and computeCell. You should move these

methods from `RefactorLife` to `TestArrayWorld`; they should become non-static, private methods and operate directly on the `cells` field. For example, the method `countNeighbours` should have the following prototype inside `TestArrayWorld`:

```
private int countNeighbours(int x, int y)
```

Testing your program

When you've finished refactoring your code, your implementation of `RefactorLife` should behave just as `LoaderLife` did last week. For example, the following should work:

```
crsid@machine:~>java -cp /home/crsid/world.jar:. \
uk/ac/cam/crsid/tick6/RefactorLife \
http://www.cl.cam.ac.uk/teaching/0809/ProgJava/life.txt 0
-
________
___#____
_#_#____
__##____
________
________
________
________
```

Pressing **Enter** should show successive generations of the Game of Life. Typing **q** followed by **Enter** should quit the application as before.

Implementing the World interface again

In the last section you produced an implementation of the interface `World` which used arrays to store the state of the world. In this section you will write a second class called `TestPackedWorld` to use a `long` integer to store the state of the world, just as you did in Tick 3.

6. Create a new class called `TestPackedWorld` by copying across the code you wrote for `TestArrayWorld` into `TestPackedWorld.java` and updating the name of the class appropriately.
7. Update the type of `cells` from `boolean[][]` to `long`.
8. Update the names of the constructors from `TestArrayWorld` to `TestPackedWorld`. Replace the constructor which takes a width and height with a constructor which takes no arguments, and modify the body of the constructors so the values of the fields `width` and `height` are always set to eight; also make sure the field `cells` is initialised to the value zero. The constructor which takes an argument of type `TestArrayWorld` should take a single argument of type `TestPackedWorld` instead.
9. Update the body of the methods `getCell` and and `setCell` so that they get or set the appropriate bits in `cells`. You should look at contents of the file `TinyLife.java` which you wrote for Tick 3 for help, and copy across your implementation of `PackedLong.java` to get and set bits in a `long` integer.
10. Update any other methods in `TestPackedWorld` which relied on `cells` being of type `boolean[][]`.

11. Update all uses of `TestArrayWorld` to `TestPackedWorld` in the two methods called `nextGeneration`.

12. Test your implementation of `TestPackedWorld` by replacing all uses of `TestArrayWorld` with `TestPackedWorld` in `RefactorLife`. Make sure that `RefactorLife` still executes correctly for worlds of size 8-by-8.

In order to use your latest implementation of `RefactorLife` you simply replaced all instances of `TestArrayWorld` with `TestPackedWorld`, but this now precludes the use of Java arrays to store the state of the world. A better solution is to accept an argument on the command line which instructs the program to use a particular implementation—arrays or `long`—as the underlying storage mechanism.

The approach you should use is to modify `RefactorLife` to accept an option on the command line to specify the underlying storage architecture used. If the user types "`--array`" as the first argument, then the storage architecture used should be a Java array, using `TestArrayWorld`; if the user types "`--long`" as the first argument then the storage used to record the state of the world should be `TestPackedWorld`. If no switch is specified, then your implementation of `RefactorLife` should default to using Java arrays to store the state of the world. Here is an example invocation which requests the use of Java arrays using the "`--array`" switch:

```
crsid@machine:~>java -cp /home/crsid/world.jar:. \
uk/ac/cam/crsid/tick6/RefactorLife --array \
http://www.cl.cam.ac.uk/teaching/0809/ProgJava/life.txt 0
```

Here is an example invocation of which requests the use of a `long` integer to store the state of the world:

```
crsid@machine:~>java -cp /home/crsid/world.jar:. \
uk/ac/cam/crsid/tick6/RefactorLife --long \
http://www.cl.cam.ac.uk/teaching/0809/ProgJava/life.txt 0
```

Here is an example invocation which specifies no storage type (your code should default to using Java arrays):

```
crsid@machine:~>java -cp /home/crsid/world.jar:. \
uk/ac/cam/crsid/tick6/RefactorLife \
http://www.cl.cam.ac.uk/teaching/0809/ProgJava/life.txt 0
```

Some further hints on how to complete this question appear in the Workbook immediately after this question box.

13. Extend your implementation of `RefactorLife` to accept an optional "switch" on the command line to enable the state of the world to be stored in an array or in a `long` integer. You should print a descriptive error message if an optional argument of an unknown type is provided.

You may recall from lectures that you cannot compare strings for equality as follows:

```
String first = new String("a");
String second = new String("a");
if (first == second) { //does not perform String equality
 System.out.println("This is not printed");
}
else {
 System.out.println("This is printed!");
}
```

this is because the equality operator for variables which reference Java objects (e.g. `first` and `second` above) returns `true` only if the two references point to the same underlying object. In the above code snippet, `first` and `second` are variables which reference two *separate* instances of a Java String which contain the same character sequence.

To check whether the *contents* of two String objects contain the same characters you should use the `equals` method of the String object:

```
if (first.equals(second)) {
 System.out.println("This is printed");
}
else {
 System.out.println("This is not printed!");
}
```

All variables which reference Java objects contain an `equals` method which you should use to compare two references for equality.

When you write your improved implementation of `RefactorLife`, you should declare a variable of type `World` to reference an instance of either `TestArrayWorld` or `TestPackedWorld` and use if-else statements to update the reference to point to the correct instance. Here is a skeleton piece of code:

```
public static main(String[] args) {

 String worldType = args.length == 3 ? args[0] : "--array";

 //TODO: initialise other variables here and interpret format string
 //      which defines the size and state of the world

 World world = null;
 if (worldType.equals("--array")) {
  world = new TestArrayWorld(/* TODO */);
 } else if (worldType.equals("--long") {
  world = new TestPackedWorld(/* TODO */);
 } else {
   //TODO: print a descriptive error message
   return;
 }

 //TODO: use the "world" variable to display the state of the world
 //      on the terminal using the "print" method
}
```

Code inheritance

Your implementation of `TestArrayWorld` and `TestPackedWorld` share substantial amounts of identical code. This is bad from a software engineering point of view because you have implemented the

same bugs twice, and you'll have to fix them in two places. When you need to add new features you will have to add them in two places too. If you add more methods of storing the state of the world by creating new classes which implement `World`, you'll then have three or more places to update with new features (or bug fixes)!

One solution to this is to use *code inheritance* to collect together the shared code in one place. Unlike interfaces, inheritance allows the programmer to share code between classes. Shared pieces of code are placed in the *parent* class, and this basic implementation is then *extended* by a *child* class. The child classes then automatically inherit any methods or fields provided in the parent class.

A parent class might not provide an implementation of all the methods which are expected to appear in the child classes. In which case, the parent class is an *abstract* class; since not all the methods have an implementation, it is not possible to create an instance of an abstract class. Next is an example abstract class which collects together all the shared code which is found in your implementation of `TestArrayWorld` and `TestPackedWorld`. A full explanation of this code is given in subsequent subsections of this Workbook.

```
package uk.ac.cam.crsid.tick6;

import java.awt.Color;
//TODO: insert other appropriate "import" statements here

public abstract class WorldImpl implements World {

 private int width;
 private int height;
 private int generation;

 protected WorldImpl(int width, int height) {
  this.width = width;
  this.height = height;
  this.generation = 0;
 }

 protected WorldImpl(WorldImpl prev) {
  this.width = prev.width;
  this.height = prev.height;
  this.generation = prev.generation + 1;
 }

 public int getWidth() { return this.width; }

 public int getHeight() { return this.height; }

 public int getGeneration() { return this.generation; }

 public int getPopulation() { return 0; }

 protected String getCellAsString(int x,int y) {
  return getCell(x,y) ? "#" : "_";
 }

 protected Color getCellAsColour(int x,int y) {
  return getCell(x,y) ? Color.BLACK : Color.WHITE;
 }
```

```
public void draw(Graphics g,int width, int height) {
 int worldWidth = getWidth();
 int worldHeight = getHeight();

 double xscale = (double)width/(double)worldWidth;
 double yscale = (double)height/(double)worldHeight;

 for(int x=0;x<worldWidth;++x) {
  for(int y=0;y<worldHeight;++y) {
   int xpos = (int)(x*xscale);
   int ypos = (int)(y*yscale);
   int nextx = (int)((x+1)*xscale);
   int nexty = (int)((y+1)*yscale);

   if (g.hitClip(xpos,ypos,nextx-xpos,nexty-ypos)) {
    g.setColor(getCellAsColour(x, y));
    g.fillRect(xpos,ypos,nextx-xpos,nexty-ypos);
   }
  }
 }
}

//TODO: Complete here in parent
public World nextGeneration(int log2StepSize) {
 //Remember to call nextGeneration 2^log2StepSize times
}

//TODO: Complete here in parent
public void print(Writer w) {
 //Use getCellAsString to get text representation of the cell
}

//TODO: Complete here in parent
protected int countNeighbours(int x, int y) {
 //Compute the number of live neighbours
}

//TODO: Complete here in parent
protected boolean computeCell(int x, int y) {
 //Compute whether this cell is alive or dead in the next generation
 //using "countNeighbours"
}

// Will be implemented by child class. Return true if cell (x,y) is alive.
public abstract boolean getCell(int x,int y);

// Will be implemented by child class. Set a cell to be live or dead.
public abstract void setCell(int x, int y, boolean alive);

// Will be implemented by child class. Step forward one generation.
protected abstract WorldImpl nextGeneration();
}
```

Note that the class `WorldImpl` has no method of storing the state of the world. This task is left to child classes which extend `WorldImpl`.

Keyword: abstract

A few new keywords have been introduced in this class definition. The keyword `abstract` prefixes both the keyword `class` and several methods. Just as you saw earlier with interfaces, abstract methods contain no body and instead are terminated with a semi-colon (`;`).

Keywords: private, protected and public

Methods and fields are prefixed with one of three *access modifiers*. Fields or methods which are marked `private` cannot be accessed by any other class (not even children); those marked `protected` can only be accessed within the class or any child (or any subsequent child of the child); those marked with `public` are accessible from any class. Finally, if a method or field does not have any access modifier, it is accessible by any class in the same package. You should use access modifiers to minimise access to fields and methods as much as possible in order to control complexity and therefore reduce the chance for unintentional interdependence between classes; this should help reduce bugs and make your code more legible to others (and yourself when you return to read it after a few months).

Keyword: extends

To create a subclass of `WorldImpl` you use the `extends` keyword. For example, to create a new child class of `WorldImpl` called `ArrayWorld` you should write the following:

```
public class ArrayWorld extends WorldImpl {
...
}
```

Keyword: super

Unlike interfaces, a class can only inherit from a *single* parent in Java. All the methods of the parent class are inherited by the child *with the exception of the constructors*. If a parent class has no default constructor (i.e. it has no constructor which takes zero arguments) then the child class *must* provide a constructor; in addition the constructor provided in the child class *must* call the constructor of the parent class. For example, the class `WorldImpl` does not have a default constructor. Consequently `ArrayWorld` as declared above must provide at least one constructor and inside it call a constructor for the parent class. This can be done as follows:

```
public class ArrayWorld extends WorldImpl {

 public ArrayWorld(int width, int height) {
  super(width,height);
  //possibly other constructor stuff here
 }
 ...
}
```

In this example the special method `super` is used to call a constructor in the parent class, in this case the constructor in `WorldImpl` which takes two `int` arguments. The call to `super` must be the first statement found inside the body of the constructor. If a child has a method body provided for every abstract method found in a parent class, then the child class is no longer an abstract class, and concrete instances of it can be created. For example:

```
World world = new ArrayWorld(8,8);
```

is possible if `ArrayWorld` provides implementations for the three abstract methods found in `WorldImpl`.

Keyword: this

The existence of a concrete class such as ArrayWorld allows the last important keyword introduced in the class WorldImpl to be discussed. The keyword is this, which is a read-only variable which references the current object instance on which the method has been called. For example, in the code snippet above, the variable world references a new instance of ArrayWorld. Inside the body of the call to the constructor of the parent class (WorldImpl), this references the new instance of ArrayWorld under construction. Therefore this.width references the *field* called width found in the class definition of WorldImpl, whereas width references the local variable of the same name which is the first argument to the constructor.

The keyword this is optional in many cases. If you refer back to the constructor for TestArrayWorld you will see that is has been omitted. This is because the arguments of the constructor are called w and h and so the member variable width can be referred to without needing to write this.width. If the argument of a method has the same name as a member variable then then argument masks the member variable.

14. Complete the class WorldImpl by finishing the non-abstract methods marked TODO by cut 'n' pasting code from methods of the same name found in TestArrayWorld and editing appropriately. Place appropriate import statements at the top of the source file and check you have done so by compiling WorldImpl.

15. Create a new class called ArrayWorld which extends WorldImpl. The class should provide a suitable implementation for each abstract method declared in the parent class so that ArrayWorld may be instantiated and is functionally identical to TestArrayWorld. You will need to provide a private field of type boolean[][] called cells to store the state of the world. In addition you should provide two constructors: one should take two int arguments specifying the width and height of the world, and one which takes a reference to an existing ArrayWorld; both constructors should call a constructor of the parent class using the special call super and then initialise cells to reference a suitable instance of a two dimensional boolean array.

16. Create a new class called PackedWorld which extends WorldImpl. The class should provide a suitable implementation for each abstract method declared in the parent class so that PackedWorld may be instantiated and is functionally identical to TestPackedWorld. You will need to provide a private field of type long called cells to store the state of the world. In addition you should provide two constructors: one should take no arguments, and one which takes a reference to an existing PackedWorld; both constructors should call a constructor of the parent class using the special call super and then initialise cells to the value zero.

17. Update RefactorLife to use your implementations of ArrayWorld and PackedWorld instead of TestArrayWorld and TestPackedWorld. Test your latest version of RefactorLife by supplying a variety of format strings which describe particular starting worlds in the Game of Life.

Benefits from World interface

Your implementation of ArrayWorld and PackedWorld both extend the abstract class WorldImpl which in turn implements the interface World. Therefore both ArrayWorld and PackedWorld are of type World. Consequently, you can use the graphics display class inside world.jar which you downloaded at the start of this workbook. To use the graphical display component, you first need to import the class by typing "import uk.ac.cam.acr31.life.WorldViewer;" at the top of RefactorLife. You can create a new instance of the WorldViewer class as follows:

```
WorldViewer viewer = new WorldViewer();
```

To use an instance of `WorldViewer` to display a `World` you can use the `show` method. For example, if you have a variable called `world` of type `World` then you can type:

```
viewer.show(world);
```

This method can be called repeatedly to update the screen when a new generation is computed. You should destroy the `WorldViewer` instance after you have finished by calling `viewer.close()`

18. Update the `play` method (or if you've integrated it, the `main` method) of your implementation of `RefactorLife` to use the `WorldViewer` class to draw a graphical representation of the world *in addition to printing the world to the terminal*.

Note: you will still need to type **q** followed by **Enter** into the terminal window to quit the application. This is acceptable this week; you will revisit this next week when you explore event handling in graphical user interfaces.

If you completed Tick 4*, then you might like to replace the draw method in `WorldImpl` with the draw method you wrote.

Another representation of the Game of Life is to use an `int[][]` to store the number of generations since a cell was last alive. An implementation of this can be downloaded from http://www.cl.cam.ac.uk/teaching/0809/ProgJava/AgingWorld.java. Place a copy of this file in the package `uk.ac.cam.crsid.tick6` and update the package declaration in the file.

19. Modify your RefactorLife program to accept the argument `--aging` which should use the implementation of `AgingWorld` rather than `ArrayWorld` or `PackedWorld` to store the state of the world.

You should test your program before you continue with this workbook. The use of the `--aging` option switch should produce the same output as using `--array`.

Keeping a history of the number of generations since each cell was last alive is not currently exploited. It would be nice to use colour to display the number of generations since the cell was last alive. To do so, you should add the following method to your `AgingWorld` class as follows:

```
protected Color getCellAsColour(int x, int y) {
 int age = getCellAge(x, y);
 final int[] colors = new int[]
 {00000000,16711680,16717568,16724224,16731136,16738048,16744960,
  16751616,16758528,16765440,16772096,16776982,16777062,16777141};
 if (age >= colors.length) { return Color.WHITE;}
  return new Color(colors[age]);
}
```

The method `getCellAsColour` *overrides* the default implementation of `getCellAsColour` found in `WorldImpl`. When the `draw` method is called on `AgingWorld`, the `draw` method provided by `WorldImpl` will run as normal. However, the `draw` method calls `getCellAsColour` and the new implementation will be used instead of the default one in `WorldImpl`. This is depicted graphically in Figure 1, “AgingWorld overrides the `getCellAsColour` method”.

20. Modify your implementation of `AgingWorld` to include the method `getCellAsColour`. Run your implementation of `RefactorLife` with the switch "`--aging`". You should now see a graphical window with a colourful rendering of Conway's Game of Life.

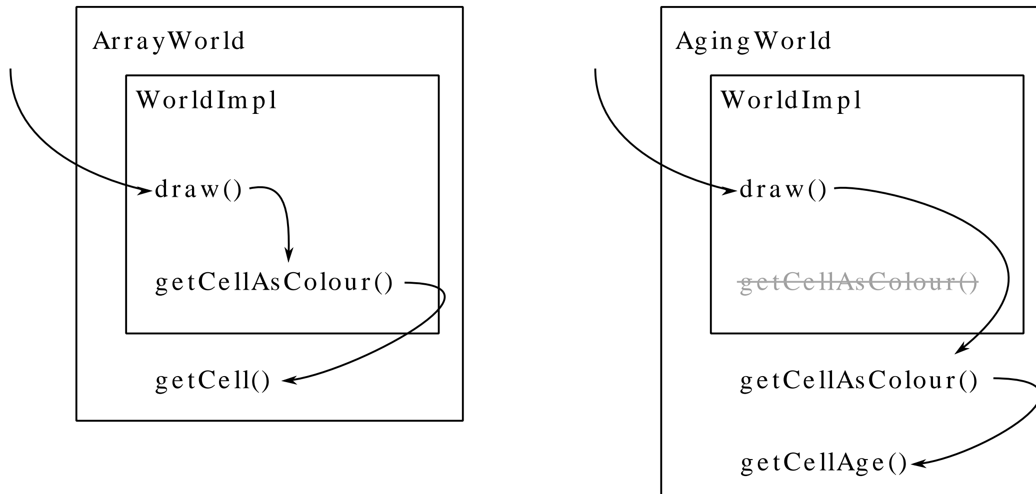

Figure 1. AgingWorld overrides the `getCellAsColour` method

Tick 6

To complete your tick you need to prepare a jar file with the contents of all the classes you have written in this workbook and email it to `ticks1a-java@cl.cam.ac.uk`. Your jar file should contain:

```
uk/ac/cam/crsid/tick6/RefactorLife.java
uk/ac/cam/crsid/tick6/RefactorLife.class
uk/ac/cam/crsid/tick6/WorldImpl.java
uk/ac/cam/crsid/tick6/WorldImpl.class
uk/ac/cam/crsid/tick6/TestArrayWorld.java
uk/ac/cam/crsid/tick6/TestArrayWorld.class
uk/ac/cam/crsid/tick6/TestPackedWorld.java
uk/ac/cam/crsid/tick6/TestPackedWorld.class
uk/ac/cam/crsid/tick6/ArrayWorld.java
uk/ac/cam/crsid/tick6/ArrayWorld.class
uk/ac/cam/crsid/tick6/PackedWorld.java
uk/ac/cam/crsid/tick6/PackedWorld.class
uk/ac/cam/crsid/tick6/AgingWorld.java
uk/ac/cam/crsid/tick6/AgingWorld.class
uk/ac/cam/crsid/tick6/Pattern.java
uk/ac/cam/crsid/tick6/Pattern.class
uk/ac/cam/crsid/tick6/PackedLong.java
uk/ac/cam/crsid/tick6/PackedLong.class
uk/ac/cam/crsid/tick6/PatternLoader.java
uk/ac/cam/crsid/tick6/PatternLoader.class
uk/ac/cam/crsid/tick6/PatternFormatException.java
uk/ac/cam/crsid/tick6/PatternFormatException.class
```

You should be able to run your program in one of three ways:

- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.crsid.tick6.RefactorLife [url/file]`
- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.crsid.tick6.RefactorLife [url/file] [index]`
- `java -cp world.jar:crsid-tick6.jar \`
  `uk.ac.cam.crsid.tick6.RefactorLife [worldType] [url/file] [index]`