Java Tick 4
Alan Blackwell – March 2004

This tick comes from chapter 10 of the BlueJ book (the **`foxes-and-rabbits`** project).

Make your own copy of the project directory **`foxes-and-rabbits-v1`** and open it in BlueJ. The first group of exercises involves exploring the functionality and behaviour of the **`foxes-and-rabbits`** simulation, and assessing the effect of changing the simulation parameters in various ways.

> Explore the **`foxes-and-rabbits-v1`** simulation as described in exercises 10.1 – 10.19. You do not need to wait for many long simulation runs, unless you are particularly interested in the behaviour of simulation programs.

The second group of exercises involves *refactoring* the simulation code, to produce a better-designed class structure, and allow a variety of extensions.

> Identify the similarities and differences between the **`Fox`** and **`Rabbit`** classes, and create an **`Animal`** superclass as described in exercises 10.20 – 10.29. If you are unsure, at the end of these exercises, whether your work has resulted in a correctly defined **`Animal`** class, you may compare your result to the **`foxes-and-rabbits-v2`** project.
>
> Learn more about the relationship between superclasses and subclasses by working through exercises 10.30 – 10.34, then create a new kind of animal by extending **`Animal`** (10.35).
>
> Refactor your solution further, introducing an **`Actor`** class (10.36 – 10.37). Define a new type of actor that behaves differently from animals (10.38). You may implement a **`Hunter`** class as described in the BlueJ book, or invent a completely new kind of actor if you like.
>
> Implement the **`SimulatorView`** interface, **`AnimatedView`** and **`TextView`**, following the suggestions in exercises 10.39 – 10.41.

For your tick assessment, run your enhanced simulation (including the new Animal and Actor types), and capture the output produced by both kinds of **`SimulatorView`**. *Capture the window containing the* **`AnimatedView`**, *and print this using the* **`import`** *command as in the second tick. Capture the text output from* **`TextView`** *as follows: Start the emacs editor. Use the mouse to select the text in the BlueJ output window, and copy it. "Yank" (paste) the text in emacs. Save the resulting text to a temporary file. Print the temporary file using* **`lpr`**, *then delete it.*

Java Tick 5

This tick comes from chapter 11 of the BlueJ book (the **`address-book`** project).

Copy the six different projects in the address-book family from the PWF BlueJ directory. These are nested within a single directory called **`address-book-family`**. If you use the **`cp -r`** command as in previous exercises, it will copy all nested directories to your local space.

> Explore and compare the **`address-book-v1g`** and **`address-book-v1t`** projects (exercises 11.1-11.6). Investigate the circumstances in which these projects are subject to null pointer errors (11.7, 11.10, 11.11, 11.12). Contrast the approach taken in the **`address-book-v2g`** and **`address-book-v2t`** projects (11.13 – 11.15). Make a new copy of the **`address-book-v2t`** project, and modify it to provide failure information to a client (exercise 11.17).
>
> Review all the methods of the **`AddressBook`** class and add checks and throw statements for the **`IllegalArgumentException`** (11.22). Add javadoc documentation to describe exceptions thrown by its methods (11.23).
>
> Enhance the **`address-book-v3t`** project to throw and handle exceptions (11.24, 11.25). Handle checked and unchecked exceptions in different catch clauses (11.26). Define a new checked exception **`DuplicateKeyException`**, modifying the project to report and catch it wherever necessary (11.27).

For your tick assessment, print out the source code, and the javadoc documentation, for the **`AddressBook`** *class, showing your use of the* **`IllegalArgumentException`** and **`DuplicateKeyException`** **`[PTO]`**.

Investigate the file input and output in the **`address-book-io`** project (section 11.8 of the BlueJ book). Adapt what you have learned (reusing code from the **`address-book-io`** project), to enhance the **`tech-support`** project that you created in the third ticked exercise. Modify that project so that it reads key words and responses from a text file (exercise 11.29). Make a further modification so that it writes a log-file recording user input. Look for a Java library class that can be used to include the time of day with each user interaction recorded in this log file. Implement a *regression testing* facility (review pages 150-154 in the BlueJ book) that simulates user input by reading it in from a previously recorded log file, and then compares the system response to the expected response as recorded in the log file.

For your tick assessment, print out a copy of the log file that you recorded and used for regression testing, and the source code of the regression testing class.

Java Tick 6

This tick comes from chapters 7 and 9 of the BlueJ book (the **`zuul`** project).

Beware – developing this kind of adventure game can be addictive! Spend just enough time at each development step that you learn the Java point being made, but don't spend too much time creating elaborate game scenarios with large amounts of text.

Explore the two **`zuul`** projects, and think about an adventure scenario that might be more entertaining (exercises 7.1 – 7.4). Using either the original, or your own scenario if you prefer, refactor and extend the functionality of the game (exercises 7.5 – 7.18, then 7.20 – 7.26).

Implement the extensions described in exercises 7.29 – 7.34. Add a transporter room (7.38), but do this using inheritance as described in chapter 9 (exercises 9.9 – 9.11).

If you have time and interest (and in return for a starred tick), implement more of the extensions described in exercises 7.35 – 7.41.

Modify your game so that it can be executed from the command line without using BlueJ (exercises 7.45 and 7.46).

For your tick assessment, copy the text of an interaction session (or run the session from emacs shell mode), save it to a file, and print it out. Print the source code of the class used to implement your transporter room. Use emacs to create a file containing extracts of code illustrating how the transporter room is used by other classes, and print this out.

Java Tick 7

This tick is loosely based on chapters 12 and 13 of the BlueJ book (design case study).

Design a timetable system for the Cambridge Computer Science Tripos. The system should read a timetable in the format of the concise lecture list summary (see **`$CLTEACH/afb21/bluej/examples/lectlist.txt`**), and provide operations to query and modify that timetable in useful ways. Modified versions can be saved by the user in the format of the original list. Students can use the system to query their commitments at particular times, or to generate a personalised timetable, in a more readable format (perhaps HTML), listing only their own lectures.

Complete the above specification, clarifying the system behaviour in a set of use case scenarios. Analyse and design the classes using the CRC method (you may work with a partner at this stage).

Implement and test the system, first calling methods directly from the BlueJ interface, then with a command line interface running independently of BlueJ (you may like to adapt the command processor from the **`zuul`** project). For a starred tick, make a graphical user interface and run your application as an applet.

For your tick assessment, draw a UML class diagram for your design, including the sections in each class that list fields and method signatures. Print out the source code of the class you consider most interesting. Provide sample output showing the personalised timetable, and either a sequence of commands, or a screen capture showing your applet executing.