
Workbook 5

Introduction

The implementation of PatternLife you wrote last week is *brittle* in the sense that the program does not cope well when input data is malformed or missing. This week you will improve PatternLife using Java exceptions to handle erroneous or missing input data. In addition, you will learn how to read files from disk and from a website and use the retrieved data to initialise a Game of Life.

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/0809/ProgJava>

You will find the Java standard library documentation useful:

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

Managing errors with Java Exceptions

Try invoking your copy of PatternLife from last week as follows:

- `java -jar crsid-tick4.jar`
- `java -jar crsid-tick4.jar "Glider:Richard Guy:20:20:1:"`
- `java -jar crsid-tick4.jar "Glider:Richard Guy:twenty:20:1:1:010 001 111"`

What does your program print out in each of the above cases? It's likely that in each case your implementation will print out a *stack trace* which describes an error in the program. Here is a typical stack trace from a student submission:

```
crsid@machine~> java -jar crsid-tick4.jar "Glider:Richard Guy:20:20:1:"
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 5
    at uk.ac.cam.crsid.tick4.Pattern.<init>(Pattern.java:48)
    at uk.ac.cam.crsid.tick4.PatternLife.main(PatternLife.java:96)
```

In this case the input string "Glider:Richard Guy:20:20:1:" provided on the command line to the program did not conform correctly to the specification described in Workbook 4. The stack trace explains where the error in the program occurred. The first line of the stack trace explains that an exception of the type `java.lang.ArrayIndexOutOfBoundsException` occurred when the sixth element of the array was accessed. The remaining lines provide a little history of program execution which led the computer to make the array access which generated the exception. In this case, program execution was taking place at line 48 of `Pattern.java` when the error occurred; this location was reached because the method on line 48 of `Pattern.java` was invoked by the constructor on line 96 of `PatternLife.java`. The detail in the stack trace helps the programmer determine why the error occurred and provides clues on how to fix it.

The exception `java.lang.ArrayIndexOutOfBoundsException` is actually a class inside the package `java.lang`. The `java.lang` package is special in Java because, unlike class-

es in all other packages, the contents of this package are always available in a Java program. Consequently, you can write `ArrayIndexOutOfBoundsException` instead of `java.lang.ArrayIndexOutOfBoundsException`.

Take a second look at each of the errors generated by your code with the three test cases mentioned at the start of this section. Can you determine which assumptions were made by your program which led to the error occurring? In some cases you can avoid generating errors by checking inputs carefully before using them; in other cases you will need to write additional code to *catch* the error and handle it. For example, you can probably avoid an exception of the type `ArrayIndexOutOfBoundsException` by checking the `length` field of the array before accessing particular elements of the array. In contrast, exceptions of the type `NumberFormatException` need to be caught and handled appropriately.

If you need to handle an error, then you can do this by using a *try-catch* block. Consider the following example:

```
int width;
try {
    width = Integer.parseInt("twenty"); //error: not an integer value
} catch (NumberFormatException error) {
    //handle the error, perhaps by using a default:
    width = 10;
}
```

The above code attempts to convert the Java string "twenty" into a number, which fails since the contents of the string doesn't contain digits describing an integer literal. The static method `parseInt` then *throws* an exception of type `NumberFormatException` which is caught by the try-catch block. In the case above, the programmer has decided to hard-code the value of `width` to 10. In some cases, using a default value like this is satisfactory. In the case of `PatternLife`, providing a default value for `width` is not ideal because the programmer cannot know the size of the world the user wishes to simulate—this is why the format string provides the information in the first place!

In cases where no default value is sensible, the only option is to *throw* an exception, as opposed to a normal return value, back to the calling method in the hope that this method might know what to do to handle the error. Ultimately, the programmer might not know what to do at any point in the program, in which case all the programmer can do is display an error message to the user. You will explore how to throw exceptions between methods after the next exercise.

In more complex cases, you may need to handle multiple types of exception separately. You can attach multiple *catch* blocks to a single *try* block as shown in the following example:

```
try {
    //code which may generate multiple types of exception
} catch (TypeAException a) {
    //handle TypeAException here
} catch (TypeBException b) {
    //handle TypeBException here
}
```

Create a class called `Repeat` in package `uk.ac.cam.crsid.tick5` with the following contents:

```
package uk.ac.cam.crsid.tick5;

public class Repeat {
    public static void main(String[] args) {
        System.out.println(parseAndRep(args));
    }

    /*
     * Return the first string repeated by the number of times
     * specified by the integer in the second string, for example
     *
     *     parseAndRep(new String[]{"one", "3"})
     *
     * should return "one one one". Adjacent copies of the repeated
     * string should be separated by a single space.
     *
     * Return a suitable error message in a string when there are
     * not enough arguments in "args" or the second argument is
     * not a valid positive integer. For example:
     *
     * - parseAndRep(new String[]{"one"}) should return
     *   "Error: insufficient arguments"
     *
     * - parseAndRep(new String[]{"one", "five"}) should return
     *   "Error: second argument is not a positive integer"
     */
    public static String parseAndRep(String[] args) {
        //TODO
    }
}
```

1. Complete the method `parseAndRep` in the class `Repeat`, taking care to adhere to the specification provided in the comment written above the method.

You can test your implementation by passing zero, one or two arguments on the command line. Here is an example:

```
crsid@machine:~> java uk.ac.cam.crsid.tick5.Repeat "UoC" 5
UoC UoC UoC UoC UoC
crsid@machine:~>
```

The error handling you provided in the `Repeat` class above works well for the small example at hand, but passing around strings containing messages for the user is cumbersome and messy. As you have already seen for `Integer.parseInt`, Java provides a mechanism for passing exceptions (as opposed to return values) between methods. In Java terminology, we say that a method *throws* an exception. For example, the `Integer.parseInt` method throws an exception of type `NumberFormatException`.

To throw an exception you use the keyword `throw`. If the exception is thrown inside the body of a try-catch block, execution passes to the first line of the `catch` body which catches an exception of the appropriate type. If the call to `throw` is not contained within the body of a try-catch block, then the exception is propagated back to the method which invoked the current method, and so on recursively, until an enclosing try-catch block is found. If no try-catch block exists, then the java runtime halts the program and prints a stack trace, just as we saw earlier. Here is an example:

```
package uk.ac.cam.crsid.tick5;

class ExceptionTest {
    public static void main(String[] args) {
        System.out.print("C");
        try {
            a();
        } catch (Exception e) {
            System.out.print(e.getMessage());
        }
        System.out.println("A");
    }

    public static void a() throws Exception {
        System.out.print("S");
        b();
        System.out.print("J");
    }

    public static void b() throws Exception {
        System.out.print("T");
        if (1+2+3==6)
            throw new Exception("1");
        System.out.print("V");
    }
}
```

2. What does this program print out when executed? Type the program in to check your answer.

In the above example you should have noticed that methods `a` and `b` have an extra phrase `throws Exception` appended on the end of the method prototype. This phrase is required, and informs the programmer and the Java compiler that this method *may* throw an exception of type `Exception`. If you forget to type `throws Exception`, then you will get a compile error; you may like to temporarily delete the phrase from your copy of `ExceptionTest` to see the compile error.

A new exception can be defined by creating a new class and declaring that it is of type `Exception`. For example the following code snippet creates a new exception called `PatternFormatException`:

```
package uk.ac.cam.crsid.tick5;

public class PatternFormatException extends Exception {
}
```

This code should be placed in a file called `PatternFormatException.java` inside a suitable directory structure to match the package declaration, just as you would do for any other class in Java. You can place methods and fields inside `PatternFormatException`, just as you would in other Java classes. The syntax `extends Exception` indicates that `PatternFormatException` is of type `Exception`. This is an example of *inheritance* in Java; you will learn more about inheritance in Workbook 6. In this workbook we will limit use of inheritance to the creation of new types of exception as shown above.

As you saw in the example above, if you throw a `PatternFormatException` inside a method body and do not enclose the use of `throw` inside a try-catch block, you should append `throws PatternFormatException` on to the end of the method prototype. A method can throw more than one type

of exception, in which case the method prototype should include a comma separated list of exceptions, such as "throws PatternFormatException, NumberFormatException".

Java actually supports two types of exception: checked exceptions and unchecked exceptions, and some of the common exceptions in Java, such as `NumberFormatException`, are unchecked exceptions. A piece of code which may potentially throw a checked exception *must* either catch it in a try-catch block or declare that the method body may throw the exception; an unchecked exception does not need to be caught or declared thrown. When defining your own exceptions it is generally good programming practise to use checked exceptions (by inheriting from `Exception` as shown earlier), and you should do so in all cases in this course.

Your next task is to modify your implementation of Conway's Game of Life from last week to provide the user of your program with clear feedback on any errors in the format string. You should:

3. Make a copy of `Pattern.java` and `PatternLife.java` which you wrote for Tick 4 and place them inside the package `uk.ac.cam.crsid.tick5`.
4. Declare a new exception called `PatternFormatException` inside the same package.
5. Modify your implementation of `Pattern` so that the constructor for the class as well as the method `initialise` throws `PatternFormatException` when an error is discovered in the format string describing the world.
6. Catch the `PatternFormatException` thrown by either the creation of an instance of `Pattern` or by the use of the `initialise` method inside the `main` method of `PatternLife`, and print out a helpful message describing the error to the user.

The following invocations of `PatternLife` should print out helpful error messages. None should produce a stack trace.

- `java uk.ac.cam.crsid.tick5.PatternLife`
- `java uk.ac.cam.crsid.tick5.PatternLife RandomString`
- `java uk.ac.cam.crsid.tick5.PatternLife \`
`"Glider:Richard Guy:20:20:1:"`
- `java uk.ac.cam.crsid.tick5.PatternLife \`
`"Glider:Richard Guy:a:b:1:1:010 001 111"`
- `java uk.ac.cam.crsid.tick5.PatternLife \`
`"Glider:Richard Guy:20:20:one:1:010 001 111"`
- `java uk.ac.cam.crsid.tick5.PatternLife \`
`"Glider:Richard Guy:20:20:1:1:010 0a1 111"`

Do not forget to check that your program still produces the correct output if it is given a correctly formatted string.

Reading data from files and websites

In the rest of this Workbook you will improve the facilities used to load patterns in your implementation of Conway's Game of Life so that, by the end of this workbook, your program will be able to load patterns from files in the filesystem, or download them from websites. To do this we are going to investigate the Input-Output (IO) facilities available in the Java standard library. Handling input and output is a common source of errors in most programming languages because lots of things can go wrong: files might not exist, the contents of the file may be corrupt, or the network connection may disappear whilst data is being retrieved. Good IO programming requires careful checking of error conditions.

The Java IO standard library has two main methods of accessing data: Streams and Readers. Both of these mechanisms use exceptions to communicate erroneous states to the programmer using the library. A Stream is used for reading and writing sequences of binary data—examples might be images or Java class files. A Reader is used for reading and writing sequences of characters—such as text files, or in case the case of this workbook, strings which specify the state of the world in the Game of Life. In principle, sequences of characters can be read using a Stream, however character data can be saved in a variety of different formats which the programmer would then have to interpret and decode. In contrast, a Reader presents the same interface to character data regardless of the underlying format.

Start a web browser and take a look at Sun's documentation for the `Reader` class, paying particular attention to the methods defined for reading characters. For example, the method prototype `int read(char[] cbuf)` describes a method which reads data into a `char` array and may throw an `IOException` if an error occurs during the reading process; the return value indicates the number of characters read or `-1` if no more data is available. You may have noticed that the `Reader` class is an *abstract* class; the details of what an abstract class is and how to use it will be described in Workbook 6. This week it is sufficient to appreciate that an abstract class provides a specification which describes how a specific implementation of a "Reader" must behave. For example, `FileReader` provides a concrete implementation of `Reader`, and is able to read data from files in the filesystem.

Now is an appropriate point to explore how `System.out.println` works. The `System` class is part of the package `java.lang` and is therefore available by default. If you look for the class `System` in Sun's documentation, you see that it has a public static field called `out` of type `PrintStream`.¹ If you view the documentation for `PrintStream` you will see that the field `out` supports a variety of method calls including the now familiar `println` method. For completeness, the interested reader might like to explore what `System.err` and `System.in` do too.

Your final task this week is to write a new class called `PatternLoader`, which is capable of loading patterns from the disk or downloading them from a website. Create a new class with the following contents, making sure you give the class the correct filename and you place it in an appropriate directory structure:

```
package uk.ac.cam.crsid.tick5;

import java.io.Reader;
import java.io.IOException;
import java.util.List;

public class PatternLoader {

    public static List<Pattern> load(Reader r) throws IOException {
        //TODO: Complete the implementation of this method.
    }
}
```

This class introduces a number of new concepts which require further explanation. You should read the rest of this section of the workbook before completing your implementation of `PatternLoader`.

In your implementation of `PatternLoader` you will need to make use of some classes in the standard library such as `Reader` which you looked up in the documentation earlier. To save you from typing `java.io.Reader` at every point in the program when you want to refer to the `Reader` class, the code above makes use of the `import` statement. The statement `"import java.io.Reader;"` tells the compiler that all occurrences of `Reader` in the source file actually refer to `java.io.Reader`. Using the `import` statement will save you some typing, make your code more readable, and provide you with an explicit list of dependencies for the program at the top of the source file.

There is nothing special about classes defined in the standard library. For example, including

¹The astute reader will have noticed that we stated earlier that a `Reader` should be used for character data and the type of `System.out` is `PrintStream`! This is because `Reader` was not introduced into Java until version 1.1.

```
import uk.ac.cam.crsid.tick2.TestBit;
```

at the top of a Java source file would allow you to write `TestBit` to refer to your implementation of `uk.ac.cam.crsid.tick2.TestBit` you wrote for Tick 2.

You may recall from last week that a `static` method is associated with a class rather than an instance of a class. Therefore you can make use of `PatternLoader` just as you used `PackedLong` in previous weeks—as a library of useful methods which you can call without first creating an instance of class `PatternLoader`. For example, to call the `load` method from another class, you simply write `PatternLoader.load` followed by a reference to a `Reader` object inside round brackets.

The `load` method takes a single argument of type `Reader`. When the `load` method is invoked, a specific kind of `Reader` will be provided (for example, a `FileReader`). By specifying the type of the argument to `load` as `Reader` the method is agnostic to the actual type of `Reader` provided: the implementation of `load` does not need to consider where the data is coming from—it can simply read characters using the support provided by the particular instance of `Reader` provided by the calling method.

The return type of the `load` method is `List<Pattern>`. A `List` is another class from the Java standard library. A `List` records an ordered sequence of items and the main difference between a `List` and a Java array is that a list can change its size dynamically: the programmer can add or delete items to it without stating how large it should be in advance. The phrase "`<Pattern>`" is an example of something called Java generics, the details of which are beyond the scope of this course. This year, all you need to know is how to use classes which use Java generics. As you've seen already, all you need to do is provide the class you want to use inside the angle brackets (`<` and `>`). For example, `List<Pattern>` is a `List` which stores elements of type `Pattern`; you will learn more about Java generics next year.

The phrase "throws `IOException`" states that the `load` method *may* throw an exception of type `IOException`. The `IOException` class is defined as part of the Java standard library and is used to communicate that something unexpected happened whilst data was read or written. For example, if the network connection to the computer breaks whilst a Java program is downloading content from a website, then the `Reader` object may throw an `IOException`.

To complete `PatternLoader` you will need to implement the method `load`, which should read all the data available from the `Reader` object reference `r`, and create a `List<Pattern>` object. The type of the return value provides a strong hint that your implementation of the `load` method may well find several pattern strings available in the input. Therefore some method of separating patterns in the input stream is required.

A common technique for separating text data in Unix-like systems such as Linux is to look for "new line" characters, which in Java are written using the character literal `'\n'` and appear as new lines when printed. In contrast, Windows usually uses separate characters for "new line" (`'\n'`) and "carriage return" (`'\r'`) and therefore it's also common to see the two character string `"\r\n"` as a line separator. You might like to try writing a simple test program which executes:

```
System.out.println("A sentence on one line.\nThis is on a second line.");
```

and examine the output. This course will use a Unix-style line separator to place multiple patterns into a single file.

The methods provided by `Reader` do not provide a mechanism for dividing the input based on the presence of new line characters. This is because the `Reader` class provides low-level access to character data. The functionality to split on new lines is provided by `BufferedReader`; this functionality is possible with `BufferedReader` because the class caches data read internally, allowing the class to search for new line characters in its cache. If you check the documentation for `BufferedReader` you will see it provides a `readLine` method which will read a line from the underlying reader and return a reference to a `String` object containing the data, or alternatively return `null` if there are no more lines to be read. The method `readLine` will function correctly regardless of whether Unix- or Windows-style line

separators are used. You can create a reference to a `BufferedReader` object by passing an instance of the `Reader` object in as an argument to the constructor:

```
BufferedReader buff = new BufferedReader(r);
```

To complete your implementation of `load` you will also need to create an instance of `List` to save `Patterns` as you load them:

```
List<Pattern> resultList = new LinkedList<Pattern>();
```

Just as we saw earlier with the `Reader` class, the `List` class may have multiple implementations; in the case above, we use the `LinkedList` implementation. Given an instance of type `List` you can then add objects of the correct type as follows:

```
Pattern p = ....
resultList.add(p);
```

You can determine the current number of elements stored in a `List` object by using the `size` method, and retrieve elements using the `get` method; Sun's documentation contains further detail which you will need to review. There is also a special for-loop syntax for Java Collection objects such as `List` which allows you to iterate though all the elements in the list:

```
for(Pattern p: resultList) {
    //p references each element of "resultList" in order so that first time
    //round the loop, p references the first element, second time round the
    //second element, and so on. The loop terminates when "resultList" has
    //no more elements.
}
```

7. Complete your implementation of `load`. Your implementation should read in all available patterns from the `Reader` object, convert the pattern strings into `Pattern` objects, and store all valid patterns in a `List` object. Your `load` method should return the `List` object to the caller. If the `Reader` object contains no valid patterns, your implementation should return an empty `List` object.

Now add the following two methods to your implementation of `PatternLoader`:

```
public static List<Pattern> loadFromURL(String url) throws IOException {
    URL destination = new URL(url);
    URLConnection conn = destination.openConnection();
    return load(new InputStreamReader(conn.getInputStream()));
}

public static List<Pattern> loadFromDisk(String filename) throws IOException {
    return load(new FileReader(filename));
}
```

These two methods use your `load` method to load patterns from either a file on disk or a website. They do this by constructing a suitable `Reader` object and passing a reference to it to your method. Since your method is agnostic to the type of `Reader` provided, your implementation of `load` will function with data from either disk or from the web. You will need to add `import` statements to describe the location of the extra classes used inside the method bodies of `loadFromURL` and `loadFromDisk`; you can find the full names for the classes by looking them up in the Java documentation.

downloadable from the course website; if you do so `q` can be used to quit the program `less` once you have located the index of a pattern you would like to view.

Once you believe you have completed all the exercises in this workbook successfully, you should produce a jar file called `crsid-tick5.jar` with the following contents:

```
META-INF
META-INF/MANIFEST.MF
uk/ac/cam/crsid/tick5/Repeat.java
uk/ac/cam/crsid/tick5/Repeat.class
uk/ac/cam/crsid/tick5/ExceptionTest.java
uk/ac/cam/crsid/tick5/ExceptionTest.class
uk/ac/cam/crsid/tick5/Pattern.java
uk/ac/cam/crsid/tick5/Pattern.class
uk/ac/cam/crsid/tick5/PatternLoader.java
uk/ac/cam/crsid/tick5/PatternLoader.class
uk/ac/cam/crsid/tick5/PatternFormatException.java
uk/ac/cam/crsid/tick5/PatternFormatException.class
uk/ac/cam/crsid/tick5/LoaderLife.java
uk/ac/cam/crsid/tick5/LoaderLife.class
```

You should set the entry point of the jar file to `uk.ac.cam.crsid.tick5.LoaderLife` so you can execute your implementation of `LoaderLife` without explicitly specifying a class to execute. To submit your work, email your jar file to `ticks1a-java@cl.cam.ac.uk`.