Using the try-statement in returning a value in an array

Class JLiveWindow provides a GUI (shown to the right) with some int fields, into which a user is expected to type integers. In this Activity, we show the use of the try-statement to catch two kinds of errors that may be associated with using these fields.

Method getIntField of class JLiveWindow is supposed to return the value that is in field number f. The method has to watch out for two errors: integer f may not be the number of one of the windows shown and field number f may not contain an integer.

Expression

```
intFields[f].getText()
```

obtains the sequence of characters in field f. We don't explain it except to say that tntFields is an array and that method getText returns the desired String value. If f is not in the range of the array, the attempt to reference intField[f] will throw an ArrayIndexOutOfBoundsException, which must be caught. So we enclose this expression in a try-statement and catch this Exception.

```
/** Return the integer in integer field number f,
  *   or 0 if either f isn't valid or intField[f] doesn't contain an integer. */
public int getIntField(int f) {
   try {
       return Integer.parseInt(intFields[f].getText());
   }
    catch (ArrayIndexOutOfBoundsException ex) {
       return 0;
   }
}
```

According to the specification of the method, 0 should be returned in case f is out of range, so that's what the catch-block does. So, we have taken care of one kind of error.

The sequence of characters must be converted to an integer. Wrapper class Integer has a method parseInt that performs this service, so we call it and return its result. But if the sequence of characters cannot be converted to an integer, for example, if the user typed the letter 'p', method parseInt will throw a NumberFormatException, which must be caught. The specification of method parseInt tells us this.

So we add a second catch-clause to catch this Exception. According to the specification of the method, 0 should be returned in case f is out of range, but we should also alert the user to the fact that 0 will be used. And we have taken care of the second kind of error.

```
/** Return the integer in integer field number f,
  *   or 0 if either f isn't valid or intField[f] doesn't contain an integer. */
public int getIntField(int f) {
   try {
       return Integer.parseInt(intFields[f].getText());
   }
    catch (ArrayIndexOutOfBoundsException ex) {
       return 0;
   }
    catch (NumberFormatException ex) {
       return 0;
   }
}
```

Two remarks are in order. First, you now see that a try-statement may have more than one catch-clause. When an object is thrown, these catch-clauses are looked at in top-to-bottom order, and the first one that can catch the thrown object is used.

Second, the validity of parameter f could have been checked using an if-statement. Here, catching it using a try-statement leads to a simpler, more consistent method body, since the try-statement has to be used anyway to catch the second kind of object that could be thrown.