class Input
Reading Input from the Keyboard

These notes show how to use class Input for reading values typed in at the keyboard. The class brings together a collection of useful input methods that you can call from your programs. The class is from the text book Developing Java Software 3rd edition, where it is used in some of the examples and is listed in Appendix E.

Using the class you can write interactive programs that are able to input values of the following types:

- int
- long
- double
- float
- char
- String
- BigInteger
- BigDecimal

The user of your interactive program has to type a value using the correct format for that type. For example, 123 would be a valid int value but xyz would not be. Reading input is always tricky as you have to deal with all the strange things that someone may type in!

If something is typed in that cannot be recognised as a value of the type being asked for, then the program is terminated.

Note that class Input is intended for those learning to program and should not be used for other purposes.

Using class Input

To use class Input save the source code, which is listed at the end of this document, into a file called Input.java (you can use copy and paste from the online version of these notes or from the source file on the Moodle website). Note that Input is spelt using a capital 'I'. You must use the same combination of upper and lower case letters. The file should be part of your Eclipse project, so create a new empty file in the project and then copy and paste the source code into it.

If an interactive program using class Input is run within the Eclipse IDE then the Console View will act as a terminal window displaying the text being output and reading input from the keyboard. The Console View is usually displayed in the lower part of the IDE window. You may need to select the Console tab or open a new Console view to see it. Outside the IDE a program must be run from a command line, using a MS Windows Command Prompt window, or a Mac OS X or Linux terminal window. Simply open the appropriate window, change to the correct directory and use the java command (e.g., java MyInteractiveProgram).

A program that needs to do input first has to create an Input object using the Input class. Do this by having a line in your program that looks like this:

```
Input input = new Input();
```

This line must appear before you do any input and will give you a variable called input that references an Input object. The examples use the variable name input but you can use any valid and relevant name. Once you have an Input object you call its methods to input values. For example, to input an integer you would use:

```
int x = in.nextInt();
```

When this statement is executed, the program will wait for you to type in some input. The input must be finished by pressing the <return> key, otherwise nothing more will happen! Once the input has been read an attempt will be made to recognise the characters that were typed in as an integer of type int. If successful the int value will be returned and the variable x initialised. If not, then the programs terminates with the message "Input did not represent an int value".

To let someone using your program know that input is expected, it is a good idea to print a prompt asking for the input. For example:

```
System.out.print("Type a floating point number: ");
double d = in.nextDouble();
```

Note the use of print, rather than println, so that the input follows the message on the same line.

The following test program shows the use of some of the input methods provided by the class. To run this program, save the source code to a file called InputTest.java and compile and run it, having first made sure that a copy of Input.class is in the same directory.

```
public class InputTest
{
  public void inputNumbers()
  {
    Input input = new Input();
    System.out.print("Type an integer: ");
    int n = input.nextInt();
    System.out.println("Integer was: " + n);
    System.out.print("Type a long: ");
    long l = input.nextLong();
    System.out.println("Long was: " + l);
    System.out.print("Type a double: ");
    double d = input.nextDouble();
    System.out.println("Double was: " + d);
    System.out.print("Type a float: ");
    float f = input.nextFloat();
    System.out.println("float was: " + f);

    // Note this extra input needed to remove the newline
    // left after reading the float.
    input.nextLine();
    System.out.print("Type a word: ");
    String s = input.nextLine();
    System.out.println("Word was: " + s);

    System.out.print("Type a single character: ");
    char c = input.nextChar();
    System.out.println("Character was: " + c);

    input.close();
  }

  public static void main(String[] args)
  {
    new InputTest().inputNumbers();
  }
}
```

Methods named next<type> (for example, nextInt), attempt to read a value of the specified type. Any whitespace characters (space, tab, newline) before the actual value characters are ignored.

For each next<type> method there is also a hasNext<type> method, for example hasNextInt. These methods return true if the next non-whitespace characters typed in represent a value of the specified type, false otherwise. If there is no input available, calling a hasNext method will wait for the user to type something in and press <return>.

At the end of the inputNumbers method is the line input.close();. This is not really necessary here, as the program terminates without doing anything else, although you might find that the Eclipse compiler will give you a warning if the statement is missing. A warning won't stop the program compiling and running, but it is always good practice to take notice of warnings and fix the issue highlighted. The close method has the effect of releasing some I/O resources held by the Input object, so they can be reused later in the same program if other code uses terminal based I/O.

Input Buffering

Input from the keyboard is buffered in an input buffer managed by other classes in the Java class libraries. This means that the characters typed when you are prompted to enter a value are collected up and temporarily stored until the <return> key is pressed. Your program is kept waiting while you are typing, so your program is not actually doing anything. Only after the <return> key is pressed does your program carry on executing. While your program is waiting we say it is suspended or blocked.

As a result of the buffering your program receives a whole line of characters at a time, not single characters one after another as they are typed. The line of characters includes a newline character at the end — newline is the character generated by pressing <return>. When you input a value using a next method like nextInt, an attempt is made convert the characters in the buffer to a value of the right type, in the order the characters appear. For example, if the buffer contains:

```
123\n
```

(where \n is a newline) then the characters '1', '2' and '3' are read and converted to 123. Whitespace characters (spaces, tabs, newlines) before the 123 are skipped, so:

```
   123\n
```

or

```
\n123\n
```

also results in 123 being input. If there are non-digit characters that are not whitespace before the input value, the input will fail:

```
ab 123\n
```

The characters "ab" cannot be converted to an int (in base 10), so input fails and the program terminates, regardless of the characters that appear later in the buffer.

If the input buffer contains:

```
123   456\n
```

then getting an int value using nextInt will return 123, leaving the rest of the characters in the input buffer. Inputting another int will read 456 without the program waiting for the user to type more input as the buffer already contains an integer.

If a program prompts the user to type in two integers, one after another, at the first prompt the user can enter:

```
123\n
```

Then at the second prompt:

```
456\n
```

When the first int is read the \n is left in the input buffer. When the second int is read, the \n in the input buffer is skipped over as it is whitespace. The program will then wait for the input buffer to be filled again by the user typing the next integer. This highlights that the next input is always read starting with the current contents of the buffer. The program will only wait for the user to type more input when the buffer needs to be filled. If a program does not wait for input when you expected it to, then you need to take into account this behaviour when working out what is happening.

Using the nextLine method will return a String containing the entire contents of the buffer (i.e., an entire line of text). The newline character is read and discarded, leaving the buffer empty. If nextLine is called when the buffer is already empty, the program will wait until something is typed in.

The nextChar method can be used to read a single character from the buffer. Reading a newline character, returns the newline and it is not discarded as when using nextLine.

Getting Valid Input

A loop can be used to repeatedly ask for and then check the input until the user types something considered valid. This avoids the program terminating if the user enters invalid input, for example enters a word instead of a number.

The following code asks the user to type in ‘y’ or ‘n’ (short for yes or no) and keeps doing so until the user does actually type ‘y’ or ‘n’:

```
Input input = new Input();
String reply = "";
do
{
  System.out.print("Do you want to continue (y/n): ");
  reply = input.nextLine();
} while (!reply.equals("y") && !reply.equals("n"));
System.out.println("You typed: " + reply);
```

Note that the nextLine method is used to read an entire line of input. If the input contains anything other than the single characters required, the loop continues. The newline character is ignored.

The next example code asks the user to input an integer value and repeats until the user enters a sequence of digits that can be converted into a value of type int:

```
while (true) {
  System.out.print("Enter an integer: ");
   if (input.hasNextInt()) {
     int n = input.nextInt();
     System.out.println("You typed: " + n);
     break;
   }
 input.nextLine(); // Input and discard current line.
 System.out.println("You did not type an integer! Try again.");
}
```

Note the use of the hasNextInt method to check if the input buffer contents can actually be converted to an int. The nextInt method should be called only if hasNextInt returns true to avoid the program being terminated. Also note the call to the nextLine method to empty the input buffer between each attempt to read the input. Values of other types can be input using the appropriate next and hasNext methods.

Class Input

The class Input code listing follows. The class uses a number of Java features that are not covered in COMP1007, along with several library classes. At this stage an understanding of how the class works is not needed but it is well worth the challenge of looking through it and trying to work out what all the code does. Use the text book to help. Also, note the way the class has been commented. It has actually been fully commented using documentation comments, which can be automatically processed to produce online documentation in the form of web pages. Documentation comments can include HTML tags to control the formatting, so the appearance of tags like <code> and <p> is not an error.

```
import java.io.Closeable;
import java.io.InputStream;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.InputMismatchException;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Scanner;

/**
 * An iterator class to input specific data types from an input stream
 * without throwing any exceptions.  If there are any errors during us
 * of the iterator then a message is output to the standard output and
 * the program terminates.
 *
 * This is just a simple wrapper around Scanner that implements a subset
 * of the Scanner public interface.
 *
 * This class is useful for people new to Java since it allows them to
 * write programs using input without having to fully understand the
 * read / parse / handle exceptions model of the standard Java classes
 * and in particular Scanner.  So once exceptions and object chaining
 * are covered this class ought not to be used, it is definitely just an
 * "early stepping stone" utility class for initial learning.
 *
 * @author Russel Winder
 * @version 2004-12-16
*/
public class Input implements Closeable, Iterator
{
  /**
   * A reference to the associated Scanner that supplies all the actual
   * input functionality.
   *
   * This is protected and not final rather than private and final (as
   * we might have expected it to be) so that FileInput can access the
   * variable.  This is necessary because FileInput needs to capture all
   * exceptions that can happen during construction, which means that
   * the super constructor call cannot be used.  This appears to be
   * something of a misfeature of Java.
   */
  protected Scanner scanner;

  /**
   * The default constructor of an Input that assumes System.in is to
   * be the InputStream used.
   */
  public Input()
  {
    this(System.in);
  }

  /**
   * Constructor of an Input object given an InputStream object.
   */
  public Input(final InputStream in)
```

```
{
  scanner = new Scanner(in);
}

/**
 * A finalizer to ensure all files are closed if an Input object is\
 * garbage collected.
 */
public void finalize()
{
  close();
}

/**
 * Close the file when finished with it.
 */
public void close()
{
  scanner.close();
}

/**
 * @return true if there is more input, false otherwise.
 */
public boolean hasNext()
{
  boolean returnValue = false;
  try
  {
    returnValue = scanner.hasNext();
  }
  catch (IllegalStateException e)
  {
    illegalStateExceptionHandler();
  }
  return returnValue;
}

/**
 * @return the next token (sequence of characters terminated by a
 * whitespace) in the input
 * stream.
 */
public String next()
{
  String returnValue = null;
  try
  {
    returnValue = scanner.next();
  }
  catch (NoSuchElementException nsee)
  {
    noSuchElementHandler();
  }
  catch (IllegalStateException ise)
  {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * This operation is required in order to conform to Iterator<String>
   * but is not supported.  Normally an UnsupportedOperationException
   * would be thrown to indicate this situation but the whole point is
   * not to throw exceptions so this is simply a "do nothing" method.
   */
  public void remove()
  {
  }

  /**
   * NB This method currently has a mis-feature in that it returns false
   * incorrectly when there is a single end-of-line left in the file.
   *
   * @return true if there is a char to input, false otherwise.
   */
  public boolean hasNextChar()
  {
    return hasNext();
  }

  /**
   * @return the next char in the input stream.
   */
  public char nextChar()
  {
    char returnValue = '\0';
    try
    {
      returnValue = scanner.findWithinHorizon("(?s).", 1).charAt(0);
    }
    catch (IllegalArgumentException iae)
    {
      // This cannot happen as it is clear in the statement that the
      // horizon is 1 which is > 0 and this exception only happens for
      // negative horizons.
      System.exit(1);
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return true if there is an int to input, false
   * otherwise.
   */
  public boolean hasNextInt()
  {
    boolean returnValue = false;
    try
    {
      returnValue = scanner.hasNextInt();
    }
    catch (IllegalStateException e)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return the next int in the input stream assumed to be in the
   * default radix which is 10.
   */
  public int nextInt()
  {
    int returnValue = 0;
    try
    {
      returnValue = scanner.nextInt();
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("int");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @param radix the radix of the input.
   * @return the next int in the input stream using the radix
   * radix.
   */
  public int nextInt(final int radix)
  {
    int returnValue = 0;
    try
    {
      returnValue = scanner.nextInt(radix);
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("int");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return true if there is a long to input, false
   * otherwise.
   */
  public boolean hasNextLong()
  {
    boolean returnValue = false;
    try
    {
      returnValue = scanner.hasNextLong();
    }
    catch (IllegalStateException e)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return the next long in the input stream assumed to be in the
   * default radix which is 10.
   */
  public long nextLong()
  {
    long returnValue = 0;
    try
    {
      returnValue = scanner.nextLong();
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("long");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @param radix the radix of the input sequence.
   * @return the next long in the input stream using the radix
   * radix.
   */
  public long nextLong(final int radix)
  {
    long returnValue = 0;
    try
    {
      returnValue = scanner.nextLong(radix);
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("long");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return true if there is a BigInteger to input, false
   * otherwise.
   */
  public boolean hasNextBigInteger()
  {
    boolean returnValue = false;
    try
    {
      returnValue = scanner.hasNextBigInteger();
    }
    catch (IllegalStateException e)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return the next BigInteger in the input stream assumed to be in
   * the default radix which is 10.
   */
  public BigInteger nextBigInteger()
  {
    BigInteger returnValue = new BigInteger("0");
    try
    {
      returnValue = scanner.nextBigInteger();
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("BigInteger");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @param radix the radix of the input sequence.
   * @return the next BigInteger in the input stream using the radix
   * radixtrue if there is a float to input, false
   * otherwise.
   */
  public boolean hasNextFloat()
  {
    boolean returnValue = false;
    try
    {
      returnValue = scanner.hasNextFloat();
    }
    catch (IllegalStateException e)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return the next float in the input stream.
   */
  public float nextFloat()
  {
    float returnValue = 0;
    try
    {
      returnValue = scanner.nextFloat();
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("float");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }
```

```
/**
 * @return true if there is a double to input, false
 * otherwise.
 */
public boolean hasNextDouble()
{
  boolean returnValue = false;
  try
  {
    returnValue = scanner.hasNextDouble();
  }
  catch (IllegalStateException e)
  {
    illegalStateExceptionHandler();
  }
  return returnValue;
}

/**
 * @return the next double in the input stream.
 */
public double nextDouble()
{
  double returnValue = 0;
  try
  {
    returnValue = scanner.nextDouble();
  }
  catch (InputMismatchException ime)
  {
    inputMismatchExceptionHandler("double");
  }
  catch (NoSuchElementException nsee)
  {
    noSuchElementHandler();
  }
  catch (IllegalStateException ise)
  {
    illegalStateExceptionHandler();
  }
  return returnValue;
}

/**
 * @return true if there is a BigDecimal to input,
 * false otherwise.
 */
public boolean hasNextBigDecimal()
{
  boolean returnValue = false;
  try
  {
    returnValue = scanner.hasNextBigDecimal();
  }
  catch (IllegalStateException e)
```

```
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return the next BigDecimal in the input stream.
   */
  public BigDecimal nextBigDecimal()
  {
    BigDecimal returnValue = new BigDecimal("0");
    try
    {
      returnValue = scanner.nextBigDecimal();
    }
    catch (InputMismatchException ime)
    {
      inputMismatchExceptionHandler("BigDecimal");
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return true if there is more input including an end of line
   * marker, false otherwise.
   */
  public boolean hasNextLine()
  {
    boolean returnValue = false;
    try
    {
      returnValue = scanner.hasNextLine();
    }
    catch (IllegalStateException e)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * @return all the characters in the input stream up to and including
   * the next end of line marker in the input stream.
   */
  public String nextLine()
  {
    String returnValue = null;
    try
    {
      returnValue = scanner.nextLine();
    }
    catch (NoSuchElementException nsee)
    {
      noSuchElementHandler();
    }
    catch (IllegalStateException ise)
    {
      illegalStateExceptionHandler();
    }
    return returnValue;
  }

  /**
   * The method to handle an IllegalStateException.
   */
  private void illegalStateExceptionHandler()
  {
    System.err.println("Input has been closed.");
    System.exit(1);
  }

  /**
   * The method to handle an InputMismatchException.
   */
  private void inputMismatchExceptionHandler(final String type)
  {
    System.err.println("Input did not represent " +
      (type.equals("int") ? "an" : "a") + " " + type + " value.");
    System.exit(1);
  }

  /**
   * The method to handle an NoSuchElementException.
   */
  private void noSuchElementHandler()
  {
    System.err.println("No input to be read.");
    System.exit(1);
  }
}
```