```
import java.util.Scanner;

/**
 * Driver class for CS2514 notes for lecture 2.
 *
 * @author M.R.C. van Dongen
 * @version 0.00
 * @date 2017/02/20
 */
public class Main {
    private static final String PROMPT = "Enter you comment: ";
    private static final String ERROR_MESSAGE = "Eejit";

    public static void main( final String[] args ) {
        System.err.print( PROMPT );

        final Scanner scanner = new Scanner( System.in );
        final String comment = scanner.next( );
        final Note note = Note.createNote( comment );

        if (note == null) {
            System.err.println( ERROR_MESSAGE );
        } else {
            note.println( );
        }
    }
}
```

```
/**
 * Class for preparing a note with a <EM>short</EM> comment
 *
 * @author M.R.C. van Dongen
 * @version 0.0
 *
 * @note Lecture notes for lecture 2.
 */

// $ CS2514 comments---like this one--- are tagged with a @$@ sign. They
// $ provide additional explanations. They should _not_ be added to
// $ production code because every Java programmer should know this already.
// $ I only added these comments for you, to point out some things about
// $ the code.

// $ This class has some stylistic errors. Future lectures will resolve
// $ these problems.

// $ You should use meaningful identifiers.
// $ Lines should not exceed 76 characters (or so). Longer lines may not
// $ print properly, so this is an important rule.
// $  Method bodies should be short.

// $ This is a one line comment.
// $ Multi-line comments start with @/*@ and end with @*/@.

// $ The comment at the top of the class is a JavaDoc comment with JavaDoc
// $ tags for the $ author, version, ... You should include JavaDoc comments
// $ for the class, the constructors, the your methods. They are useful
// $ because they force you to document the purpose of your code and because
// $ you can automatically turn them into HTML.
// $  The first line of a class JavaDoc should be a one-line description
// $ of the purpose of the class. I want you to write such one-line
// $ descriptions because it will force you to think about the purpose of
// $ the class and how to describe it to others.
// $  As demonstrated on the first line, JavaDoc also allows some HTML
// $ markup. HTML markup in normal comments is allowed but it doesn't make
// $ sense.
public class Note {
    // $ Notice the extra level of indentation in the class: 4 spaces, not
    // $ a tab (tab characters are no-nos). Everything ''inside''
    // $ something should be indented by the same amount of charaters. It's
    // $ for clarity.

    // $ The following is a declaration of a constant class attribute: we
    // $ only need one of them.
    // $  Class attribute declarations should go to the top of the class.
    // $ The attribute is @final@ because its value should not change.
    // Maximum length of a @Note@.
    private static final int MAX_LENGTH = 40;
    // $ The following is an instance attribute because each instance
    // $ should have its own comment.
    // $  Instance attribute declarations should also go to the top of the
    // $ class. I suggest you put them below the class instance attribute
    // $ declarations, so you can easily find them.
    // The comment of a @Note@ instance.
    private final String comment;

    // $ The following is a constructor. You can recognise them by the
    // $ fact that their name is the same as the class and by the fact
    // $ they don't have a return type.
    // $  In the following we make the constructor @private@, so it can't
```



```
    // $ be called by class users. Most people allow direct access to the
    // $ constructor but arguably this is a bad idea because that way we
    // $ can't change the API for constructing instances of the class.
    // $  Further on, we provide a @public@ factory method for creating
    // $ instances of the class.
    // $  Notice the constructor is preceded by a method JavaDoc with
    // $ a one-line description of the constructor's purpose and tags for
    // $ the constructor's parameter and ''return'' value.
    /**
     * Main constructor.
     *
     * @param comment The comment for the new instance.
     * @return An instance of this class.
     */
    private Note( final String comment ) {
        // $ Notice the extra level of indentation.
        this.comment = comment;
    }

    // $ The following is a class (@static@) method. Inside this class
    // $ you can call using @createNote( )@ or @Note.createNote( )@,
    // $ which is clearer.
    // $  Notice this method is also preceded by a JavaDoc with
    // $ a one-line description of the constructor's purpose and tags for
    // $ the mthod's parameter and its return value.
    /**
     * Construct an instance of this class.
     *
     * @param comment The comment for the new instance.
     * @return An instance of this class; @null@ if the comment is too long
     */
    public static Note createNote( final String comment ) {
        // $ In CS2514 we only have one exit point per function. This is
        // $ a contract, which you have to stick to. Following this contrac
        // $ will make it easier to reason about your code.
        // $  Since we can have only one @return@ statement, we use a
        // $ @final@ variable for the return value. We assign it in the
        // $ body of the function and return it at the end.
        final Note result;

        if (comment.length( ) > MAX_LENGTH) {
            result = null;
        } else {
            result = new Note( comment );
        }

        return result;
    }

    // $ The following is an instance method (it doesn't have @static@
    // $ in its signature).
    /**
     * Get the comment of the current instance.
     *
     * @return The comment of this instance.
     */
    public String getNote( ) {
        // $ We could have also written @return this.comment;@.
        return comment;
    }

    /**
```

```
     * Convert this instance to @String@.
     *
     * @return The @String@ representation of this instance.
     */
    @Override
    public String toString( ) {
        return getNote( );
    }

    /**
     * Print the comment of this instance.
     */
    public void print( ) {
        System.out.print( getNote( ) );
    }

    /**
     * Print the comment of this instance followed by a newline.
     */
    public void println( ) {
        print( );
        System.out.println( );
    }

    // $ This is the end of the class. Notice it is easy to recognise
    // $ because its closing brace (curly bracket) is underneath the
    // $ letter _c_ that starts the class definition. If you use proper
    // $ indentation, you can recognise the structure of your classes,
    // $ your methods, and your statements, by their relative level of
    // $ indentation.
}
```

Call-by-value mechanism.

A method declaration has formal parameters: each has a type and a name. A method call has _actual_ parameters: they are expressions/values. (An actual parameter may sometimes look like a name because it's a variable but it really is an expression.)

The following is how Java evaluates a method call. For simplicty we assume it's a class method call.

To call a method with n parameters:

- Create a fresh variable for each formal/actual parameter of the method call. These variables are usually stored on top of a stack.
- Evaluate the actual parameters from left to right and assign their calues to the temporary variables.
- Enter the body of the method. If there's a reference to a formal parameter in the body, use ''its'' current temporary variable.
- If there's a return value, you substitute it for the method call.
- Upon return, delete the most-recently created temporary variables for the method's formal parameters.