The BlueJ Tutorial

Version 2.0.1

for BlueJ Version 2.0.x

Michael Kölling
Mærsk Institute
University of Southern Denmark

Contents

1 Foreword

1.1 About BlueJ

This tutorial is an introduction to using the BlueJ programming environment. BlueJ is a Java™ development environment specifically designed for teaching at an introductory level. It was designed and implemented by the BlueJ team at Deakin University, Melbourne, Australia, and the University of Kent at Canterbury, UK.

More information about BlueJ is available at http://www.bluej.org.

1.2 Scope and audience

This tutorial is aimed at people wanting to familiarize themselves with the capabilities of the environment. It does not explain design decisions underlying the construction of the environment or the research issues behind it.

This tutorial is not intended to teach Java. Beginners of Java programming are advised to also study an introductory Java textbook or follow a Java course.

This is not a comprehensive environment reference manual. Many details are left out – emphasis is on a brief and concise introduction rather than on complete coverage of features. For a more detailed reference, see *The BlueJ Environment Reference Manual*, available from the BlueJ web site (www.bluej.org).

Every section starts with a one-line summary sentence. This allows users already familiar with parts of the system to decide whether they want to read or skip each particular section. Section 11 repeats just the summary lines as a quick reference.

1.3 Copyright, licensing and redistribution

1.4 Feedback

Comments, questions, corrections, criticisms and any other kind of feedback concerning the BlueJ system or this tutorial are very welcome and actively encouraged. Please mail to Michael Kölling (mik@mip.sdu.dk).

2 Installation

BlueJ is distributed in three different formats: one for Windows systems, one for MacOS, and one for all other systems. Installing it is quite straightforward.

Prerequisites

You must have J2SE v1.4 (a.k.a. JDK 1.4) or later installed on your system to use BlueJ. Generally, updating to the latest stable (non-beta) Java release is recommended. If you do not have JDK installed you can download it from Sun's web site at http://java.sun.com/j2se /. On MacOS X, a recent J2SE version is preinstalled - you do not need to install it yourself. If you find a download page that offers "JRE" (Java Runtime Environment) and "SDK" (Software Development Kit), you must download "SDK" – the JRE is not sufficient.

2.1 Installation on Windows

The distribution file for Windows systems is called *bluejsetup-xxx.exe*, where *xxx* is a version number. For example, the BlueJ version 2.0.0 distribution is named *bluejsetup-200.exe*. You might get this file on disk, or you can download it from the BlueJ web site at http://www.bluej.org. Execute this installer.

The installer lets you select a directory to install to. It will also offer the option of installing a shortcut in the start menu and on the desktop.

After installation is finished, you will find the program *bluej.exe* in BlueJ's installation directory.

The first time you launch BlueJ, it will search for a Java system (JDK). If it finds more than one suitable Java system (e.g. you have JDK 1.4.2 and JDK 1.5.0 installed), a dialog will let you select which one to use. If it does not find one, you will be asked to locate it yourself (this can happen when a JDK system has been installed, but the corresponding registry entries have been removed).

The BlueJ installer also installs a program called *vmselect.exe*. Using this program, you can later change which Java version BlueJ uses. Execute *vmselect* to start BlueJ with a different Java version.

The choice of JDK is stored for each BlueJ version. If you have different versions of BlueJ installed, you can use one version of BlueJ with JDK 1.4.2 and another BlueJ version with JDK 1.5. Changing the Java version for BlueJ will make this change for all BlueJ installations of the same version for the same user.

2.2 Installation on Macintosh

Please note that BlueJ runs only on MacOS X.

The distribution file for MacOS is called *BlueJ-xxx.zip*, where *xxx* is a version number. For example, the BlueJ version 2.0.0 distribution is named *BlueJ-200.zip*. You might get this file on disk, or you can download it from the BlueJ web site at http://www.bluej.org.

MacOS will usually uncompress this file automatically after download. If not, double-click it to uncompress.

After uncompressing, you will have a folder named BlueJ-*xxx*. Move this folder into your Applications folder (or where-ever you would like to keep it). No further installation is necessary.

2.3 Installation on Linux/Unix and other systems

The general distribution file for is an executable jar file. It is called *bluej-xxx.jar*, where *xxx* is a version number. For example, the BlueJ version 2.0.0 distribution is named *bluej-200.jar*. You might get this file on disk, or you can download it from the BlueJ web site at http://www.bluej.org.

Run the installer by executing the following command. NOTE: For this example, I use the distribution file *bluej-200.jar* – you need to use the file name of the file you've got (with the correct version number).

```
<j2se-path>/bin/java -jar bluej-200.jar
```

<*j2se-path*> is the directory, where J2SE SDK was installed.

A window pops up, letting you choose the BlueJ installation directory and the Java version to be used to run BlueJ.

Click *Install*. After finishing, BlueJ should be installed.

2.4 Installation problems

If you have any problems, check the *Frequently Asked Questions* (FAQ) on the BlueJ web site (http://www.bluej.org/help/faq.html) and read the *How To Ask For Help* section (http://www.bluej.org/help/ask-help.html).

3 Getting started – edit / compile / execute

3.1 Starting BlueJ

On Windows and MacOS, a program named *BlueJ* is installed. Run it.

On Unix systems the installer installs a script named *bluej* in the installation directory. From a GUI interface, just double-click the file. From a command line you can start BlueJ with or without a project as an argument:

```
$ bluej
```

or

```
$ bluej examples/people
```

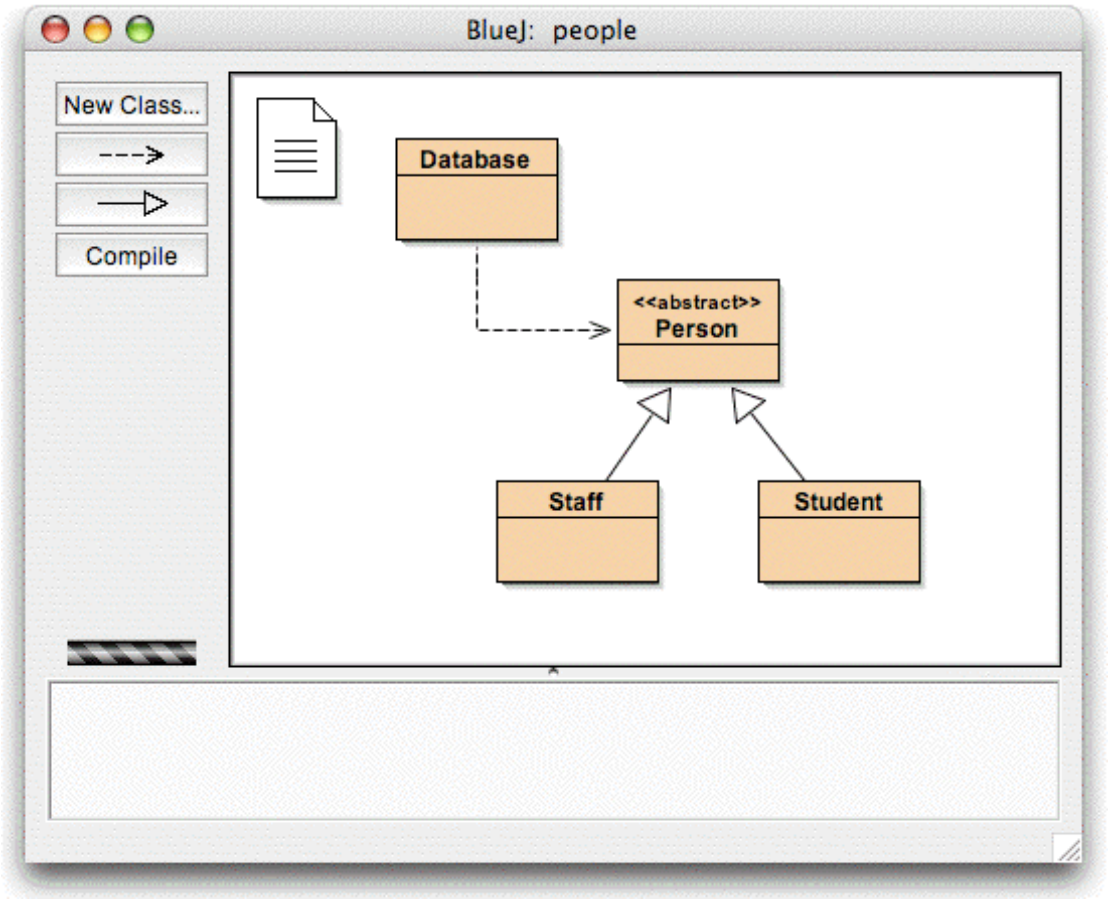

Figure 1: The BlueJ main window

3.2 Opening a project

Summary: To open a project, select Open *from the* Project *menu.*

BlueJ projects, like standard Java packages, are directories containing the files included in the project.

After starting BlueJ, use the Project – Open... menu command to select and open a project.

Some example projects are included with the standard BlueJ distribution in the *examples* directory.

For this tutorial section, open the project *people*, which is included in this directory. You can find the *examples* directory in the BlueJ home directory. After opening the project you should see something similar to the window shown in Figure 1. The window might not look exactly the same on your system, but the differences should be minor.

3.3 Creating objects

Summary: To create an object, select a constructor from the class popup menu.

One of the fundamental characteristics of BlueJ is that you cannot only execute a complete application, but you can also directly interact with single objects of any class and execute their public methods. An execution in BlueJ is usually done by creating an object and then invoking one of the object's methods. This is very helpful during development of an application – you can test classes individually as soon as they have been written. There is no need to write the complete application first.

> Side note*: Static methods can be executed directly without creating an object first. One of the static methods may be "main", so we can do the same thing that normally happens in Java applications – starting an application by just executing a static main method. We'll come back to that later. First, we'll do some other, more interesting things which cannot normally be done in Java environments.*

The squares you see in the centre part of the main window (labelled *Database*, *Person*, *Staff* and *Student*) are icons representing the classes involved in this application. You can get a menu with operations applicable to a class by clicking on the class icon with the right mouse button (Macintosh: ctrl-click[1]) (Figure 2). The operations shown are *new* operations with each of the constructors defined for this class (first) followed by some operations provided by the environment.

[1] Whenever we mention a right-click in this tutorial, Macintosh users should read this as *ctrl-click*.

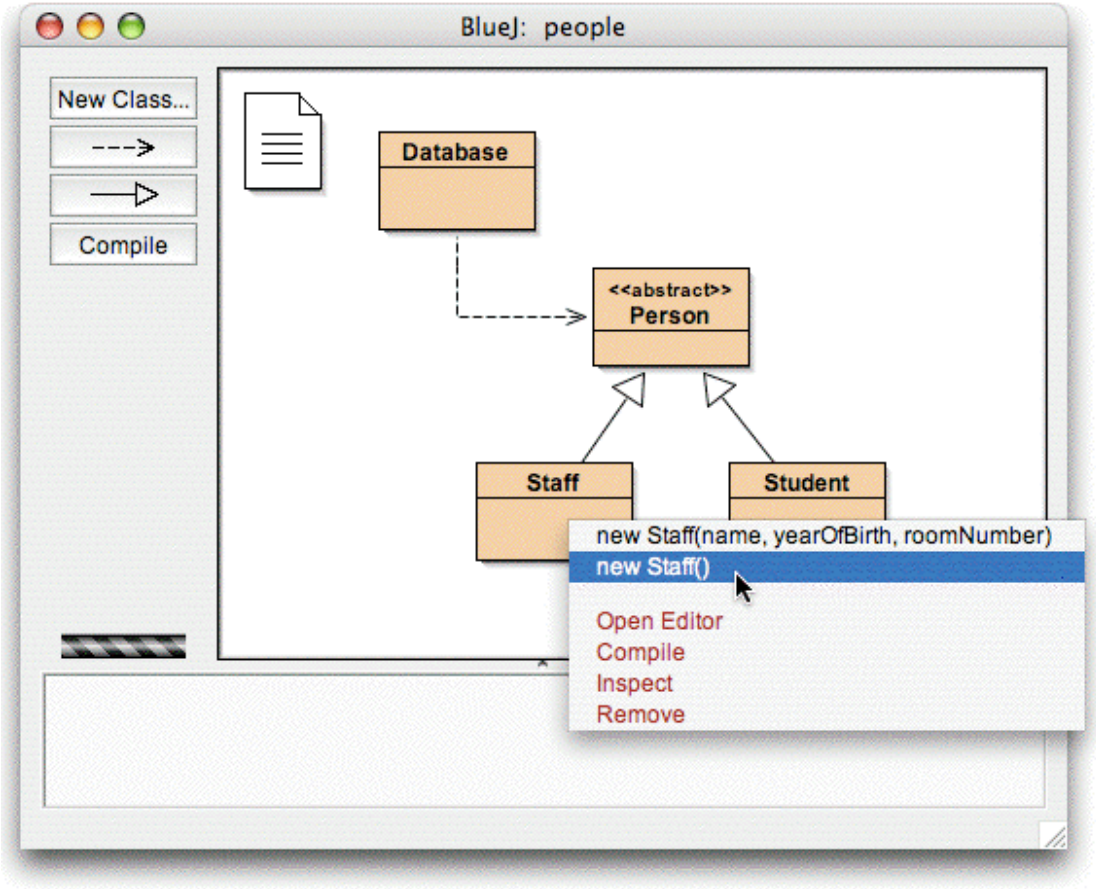

Figure 2: Class operations (popup menu)

We want to create a *Staff* object, so you should right-click the *Staff* icon (which pops up the menu shown in Figure 2). The menu shows two constructors to create a *Staff* object, one with parameters and one without. First, select the constructor without parameters. The dialogue shown in Figure 3 appears.

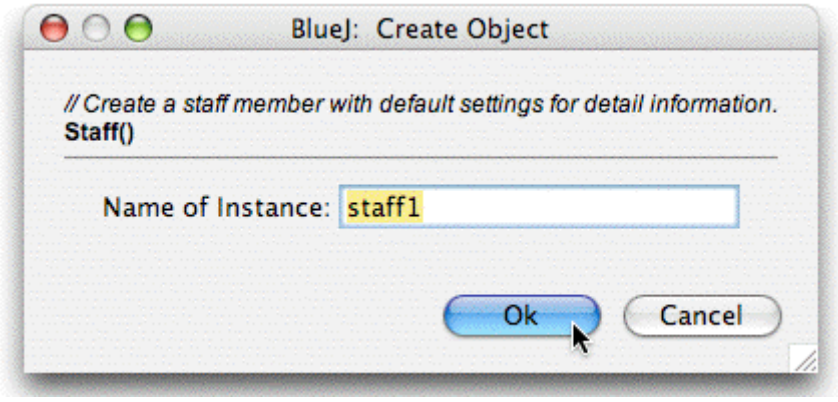

Figure 3: Object creation without parameters

This dialogue asks you for a name for the object to be created. At the same time, a default name (*staff1*) is suggested. This default name is good enough for now, so just click *OK*. A *Staff* object will be created.

Once the object has been created it is placed on the object bench (Figure 4). This is all there is to object creation: select a constructor from the class menu, execute it and you've got the object on the object bench.

Figure 4: An object on the object bench

You might have noticed that the class *Person* is labelled <<abstract>> (it is an abstract class). You will notice (if you try) that you cannot create objects of abstract classes (as the Java language specification defines).

3.4 Execution

Summary: To execute a method, select it from the object popup menu.

Now that you have created an object, you can execute its public operations. (Java calls the operations *methods*.) Click with the right mouse button on the object and a menu with object operations will pop up (Figure 5). The menu shows the methods available for this object and two special operations provided by the environment (*Inspect* and *Remove*). We will discuss those later. First, let us concentrate on the methods.

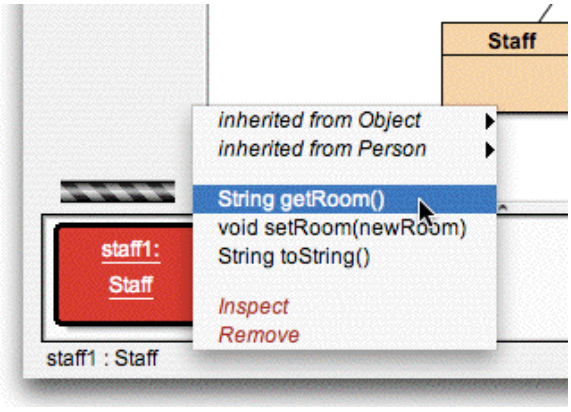

Figure 5: The object menu

You see that there are methods *setRoom* and *getRoom* which set and return the room number for this staff member. Try calling *getRoom*. Simply select it from the object's menu and it will be executed. A dialogue appears showing you the result of the call

(Figure 6). In this case the name says "(unknown room)" because we did not specify a room for this person.

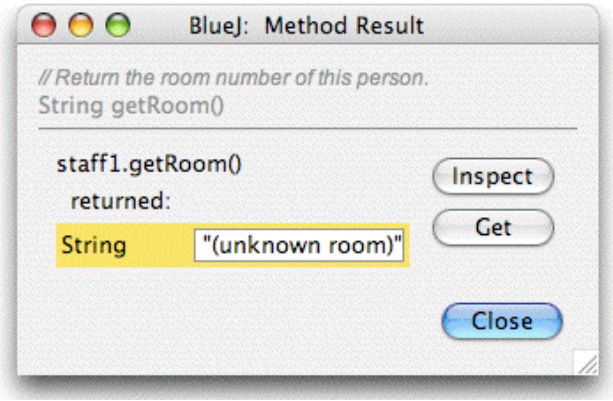

Figure 6: Display of a function result

Methods inherited from a superclass are available through a submenu. At the top of the object's popup menu there are two submenus, one for the methods inherited from *Object* and one for those from *Person* (Figure 5). You can call *Person* methods (such as *getName*) by selecting them from the submenu. Try it. You will notice that the answer is equally vague: it answers "(unknown name)", because we have not given our person a name.

Now let us try to specify a room number. This will show how to make a call that has parameters. (The calls to *getRoom* and *getName* had return values, but no parameters). Call the function *setRoom* by selecting it from the menu. A dialogue appears prompting you to enter a parameter (Figure 7).

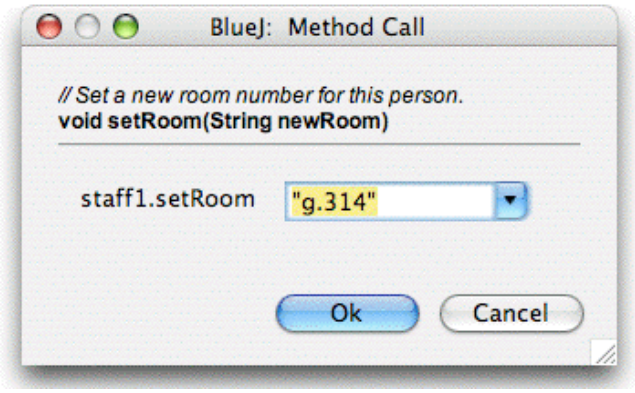

Figure 7: Function call dialogue with parameters

At the top, this dialogue shows the interface of the method to be called (including comment and signature). Below that is a text entry field where you can enter the parameter. The signature at the top tells us that one parameter of type String is expected. Enter the new room as a string (including the quotes) in the text field and click *OK*.

This is all – since this method does not return a parameter there is no result dialogue. Call *getRoom* again to check that the room really has changed.

Play around with object creation and calling of methods for a while. Try calling a constructor with arguments and call some more methods until you are familiar with these operations.

3.5 Editing a class

Summary: To edit the source of a class, double-click its class icon.

So far, we have dealt only with an object's interface. Now it's time to look inside. You can see the implementation of a class by selecting *Open Editor* from the class operations. (Reminder: right-clicking the class icon shows the class operations.) Double-clicking the class icon is a shortcut to the same function. The editor is not described in much detail in this tutorial, but it should be very straightforward to use. Details of the editor will be described separately later. For now, open the implementation of the *Staff* class. Find the implementation of the *getRoom* method. It returns, as the name suggests, the room number of the staff member. Let us change the method by adding the prefix "`room`" to the function result (so that the method returns, say, "room M.3.18" instead of just "M.3.18"). We can do this by changing the line

```
return room;
```

to

```
return "room " + room;
```

BlueJ supports full, unmodified Java, so there is nothing special about how you implement your classes.

3.6 Compilation

Summary: To compile a class, click the Compile *button in the editor. To compile a project, click the* Compile *button in the project window.*

After inserting the text (before you do anything else), check the project overview (the main window). You will notice that the class icon for the *Staff* class has changed: it is striped now. The striped appearance marks classes that have not been compiled since the last change. Back to the editor.

> Side note*: You may be wondering why the class icons were not striped when you first opened this project. This is because the classes in the people project were distributed already compiled. Often BlueJ projects are distributed uncompiled, so expect to see most class icons striped when you first open a project from now on.*

In the toolbar at the top of the editor are some buttons with frequently used functions. One of them is *Compile*. This function lets you compile this class directly from within the editor. Click the *Compile* button now. If you made no mistake, a message should

appear in the information area at the bottom of the editor notifying you that the class has been compiled. If you made a mistake that leads to a syntax error, the line of the error is highlighted and an error message is displayed in the information area. (In case your compilation worked first time, try to introduce a syntax error now – such as a missing semicolon – and compile again, just to see what it looks like).

After you have successfully compiled the class, close the editor.

> Side note*: There is no need to explicitly save the class source. Sources get automatically saved whenever it is appropriate (e.g. when the editor is closed or before a class is compiled). You can explicitly save if you like (there is a function in the editor's* Class *menu), but it is really only needed if your system is really unstable and crashes frequently and you are worried about losing your work.*

The toolbar of the project window also has a *Compile* button. This compile operation compiles the whole project. (In fact, it determines which classes need recompilation and then recompiles those classes in the right order.) Try this out by changing two or more classes (so that two or more classes appear striped in the class diagram) and then click the *Compile* button. If an error is detected in one of the compiled classes, the editor will be opened and the error location and message are displayed.

You may notice that the object bench is empty again. Objects are removed every time the implementation is changed.

3.7 Help with compiler errors

Summary: To get help for a compiler error message, click the question mark next to the error message.

Very frequently, beginning students have difficulty understanding the compiler error messages. We try to provide some help.

Open the editor again, introduce an error in the source file, and compile. An error message should be displayed in the editor's information area. On the right end of the information area a question mark appears that you can click to get some more information about this type of error (Figure 8).

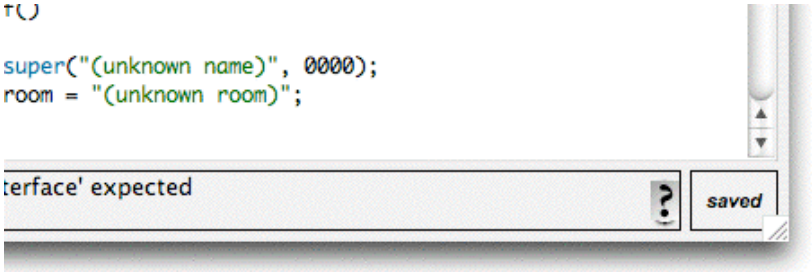

Figure 8: A compiler error and the *Help* button

At this stage, help texts are not available for all error messages. Some help text have yet to be written. But it is worth trying – many errors are already explained. The remaining ones will be written and included in a future BlueJ release.

4 Doing a bit more...

In this section, we will go through a few more things you can do in the environment. Things which are not essential, but very commonly used.

4.1 Inspection

Summary: Object inspection allows some simple debugging by showing an object's internal state.

When you executed methods of an object, you might have noticed the *Inspect* operation which is available on objects in addition to user defined methods (Figure 5). This operation allows checking of the state of the instance variables ("fields") of objects. Try creating an object with some user defined values (e.g. a *Staff* object with the constructor that takes parameters). Then select the *Inspect* from the object menu. A dialogue appears displaying the object fields, their types and their values (Figure 9).

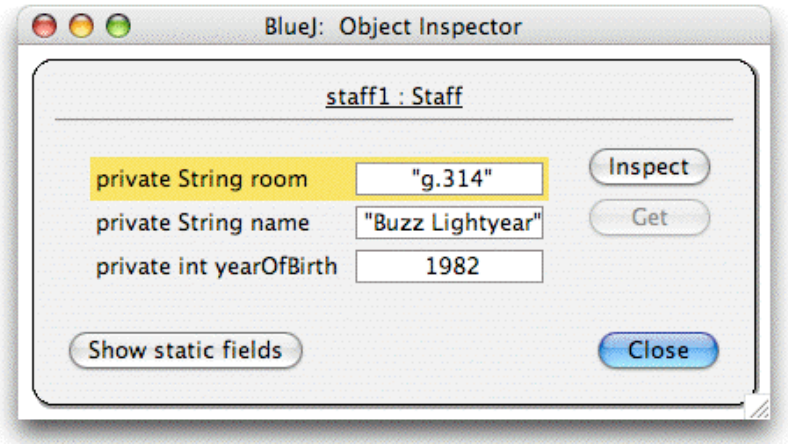

Figure 9: Inspection dialogue

Inspection is useful to quickly check whether a mutator operation (an operation that changes the state of the object) was executed correctly. Thus, inspection is a simple debugging tool.

In the *Staff* example, all fields are simple types (either non-object types or strings). The value of these types can be shown directly. You can immediately see whether the constructor has done the right assignments.

In more complex cases, the values of fields might be references to user-defined objects. To look at such an example we will use another project. Open the project *people2*, which is also included in the standard BlueJ distribution. The *people2* desktop is shown in Figure 10. As you can see, this second example has an *Address*

class in addition to the classes seen previously. One of the fields in class *Person* is of the user-defined type *Address*.

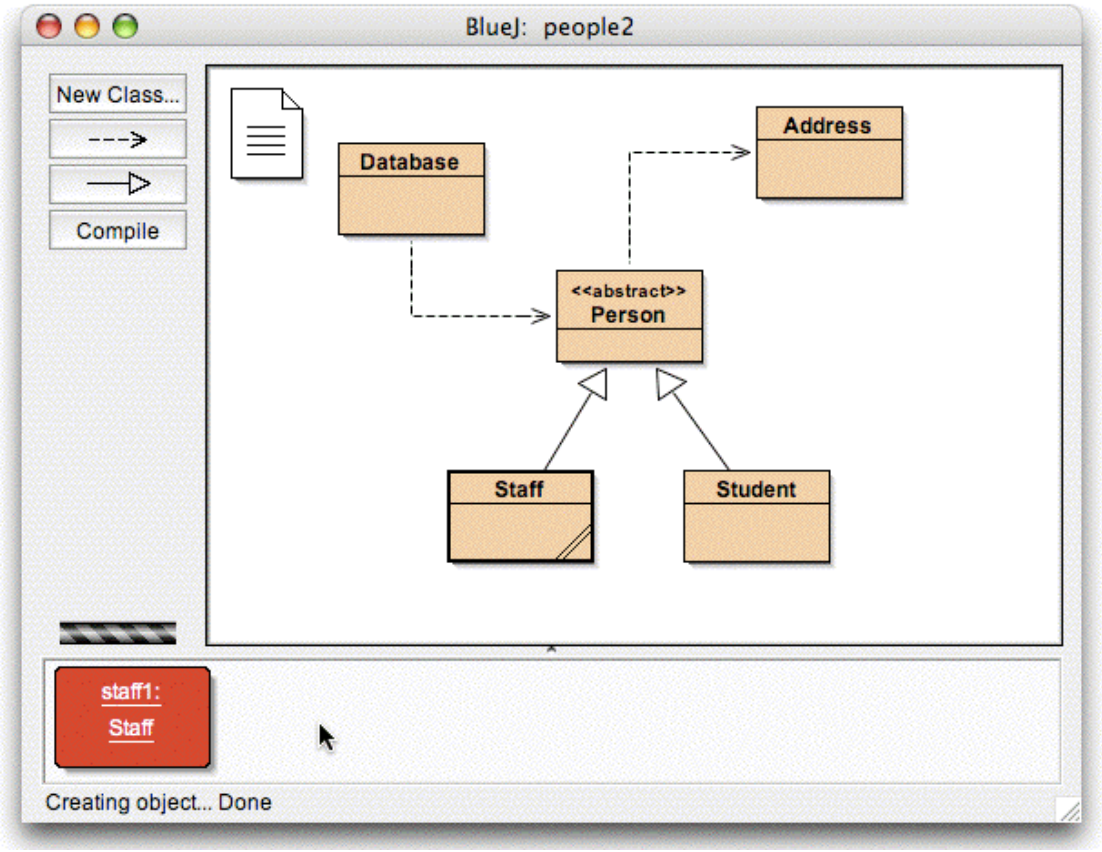

Figure 10: The *people2* project window

For the next thing that we want to try out – inspection with object fields – create a *Staff* object and then call the *setAddress* method of this object (you'll find it in the *Person* submenu). Enter an address. Internally, the *Staff* code creates an object of class *Address* and stores it in its *address* field.

Now, inspect the *Staff* object. The resulting inspection dialogue is shown in Figure 11. The fields within the *Staff* object now include *address*. As you can see, its value is shown as an arrow, which signifies a reference to another object. Since this is a complex, user-defined object, its value cannot be shown directly in this list. To examine the address further, select the *address* field in the list and click the *Inspect* button in the dialogue. (You can also double-click the *address* field.) Another inspection window is opened in turn, showing the details of the *Address* object (Figure 12).

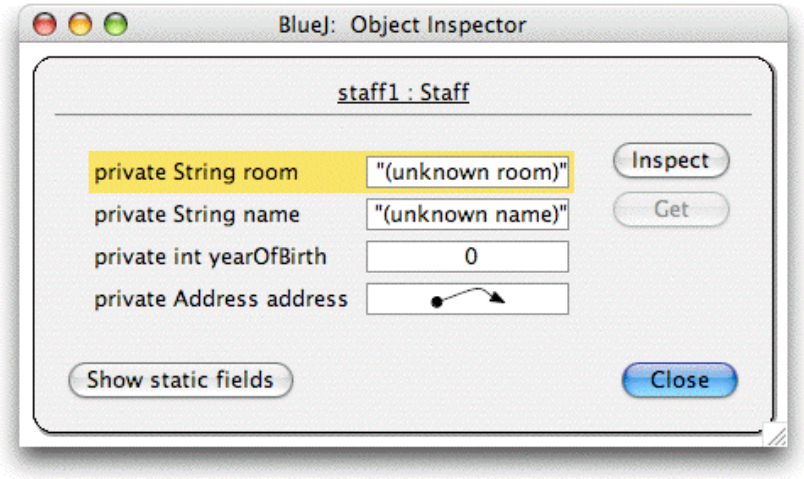

Figure 11: Inspection with object reference

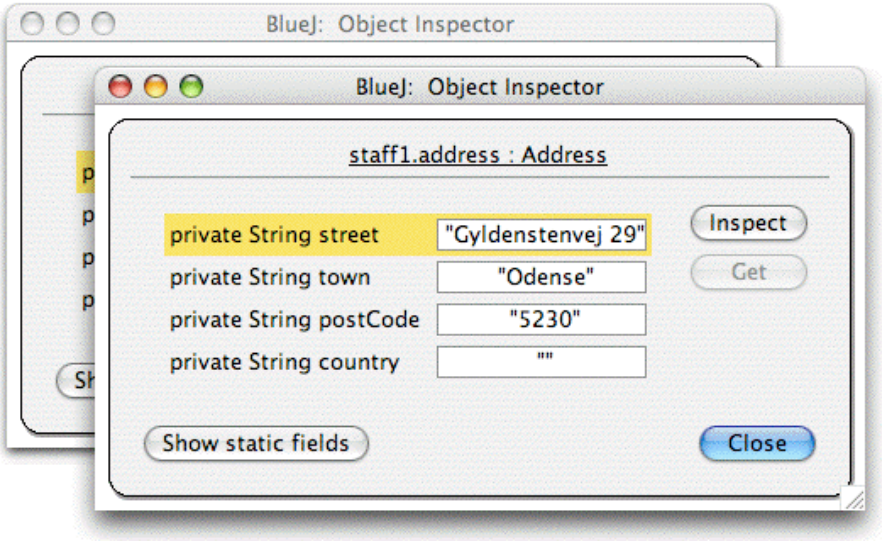

Figure 12: Inspection of internal object

If the selected field is public then, instead of clicking Inspect, you could also select the address field and click the Get button. This operation places the selected object on the object bench. There you can examine it further by making calls to its methods.

4.2 Passing objects as parameters

Summary: An object can be passed as a parameter to a method call by clicking on the object icon.

Objects can be passed as parameters to methods of other objects. Let us try an example. Create an object of class *Database*. (You will notice that the *Database* class

has only one constructor which takes no parameters, so construction of an object is straight forward.) The *Database* object has the ability to hold a list of persons. It has operations to add person objects and to display all persons currently stored. (Calling it *Database* is actually a bit of an exaggeration!)

If you don't already have a *Staff* or *Student* object on the object bench, create one of those as well. For the following, you need a *Database* object and a *Staff* or *Student* object on the object bench at the same time.

Now call the *addPerson* method of the *Database* object. The signature tells you that a parameter of type *Person* is expected. (Remember: the class *Person* is abstract, so there are no objects which are directly of type *Person*. But, because of subtyping, *Student* and *Staff* objects can be substituted for person objects. So it is legal to pass in a *Student* or *Staff* where a *Person* is expected.) To pass the object which you have on your object bench as a parameter to the call you are making, you could enter its name into the parameter field or, as a shortcut, just click on the object. This enters its name into the method call dialogue. Click *OK* and the call is made. Since there is no return value for this method, we do not immediately see a result. You can call the *listAll* method on the *Database* object to check that the operation really was performed. The *listAll* operation writes the person information to standard output. You will notice that a text terminal opens automatically to display the text.

Try this again with more than one person entered into the "database".

5 Creating a new project

This chapter takes you to a quick tour of setting up a new project.

5.1 Creating the project directory

Summary: To create a project, select New... *from the* Project *menu.*

To create a new project, select Project – New... from the menu. A file selection dialogue opens that lets you specify a name and location for the new project. Try that now. You can choose any name for your project. After you click OK, a directory will be created with the name you specified, and the main window shows the new, empty project.

5.2 Creating classes

Summary: To create a class, click the New Class *button and specify the class name.*

You can now create your classes by clicking the *New Class* button on the project tool bar. You will be asked to supply a name for the class - this name has to be a valid Java identifier.

You can also choose from four types of classes: abstract, interface, applet or "standard". This choice determines what code skeleton gets initially created for your class. You can change the type of class later by editing the source code (for example, by adding the "abstract" keyword in the code).

After creating a class, it is represented by an icon in the diagram. If it is not a standard class, the type (interface, abstract, or applet) is indicated in the class icon. When you open the editor for a new class you will notice that a default class skeleton has been created - this should make it easy to get started. The default code is syntactically correct. It can be compiled (but it doesn't do much). Try creating a few classes and compile them.

5.3 Creating dependencies

Summary: To create an arrow, click the arrow button and drag the arrow in the diagram, or just write the source code in the editor.

The class diagram shows dependencies between classes in the form of arrows. Inheritance relations ("extends" or "implements") are shown as arrows with a hollow arrow head; "uses" relations are shown as dashed arrows with an open head.

You can add dependencies either graphically (directly in the diagram) or textually in the source code. If you add an arrow graphically, the source is automatically updated; if you add the dependency in the source, the diagram is updated.

To add an arrow graphically, click the appropriate arrow button (hollow arrow for "extends" or "implements", dashed arrow for "uses") and drag the arrow from one class to the other.

Adding an inheritance arrow inserts the "extends" or "implements" definition into the class's source code (depending on whether the target is a class or an interface).

Adding a "uses" arrow does not immediately change the source (unless the target is a class from another package. In that case it generates an "import" statement, but we have not seen that yet in our examples). Having a uses arrow in the diagram pointing to a class that is not actually used in its source will generate a warning later stating that a "uses" relationship to a class was declared but the class is never used.

Adding the arrows textually is easy: just type the code as you normally would. As soon as the class is saved, the diagram is updated. (And remember: closing the editor automatically saves.)

5.4 Removing elements

Summary: To remove a class or an arrow, select the remove function from its popup menu.

To remove a class from the diagram, select the class and then select *Remove* from the *Edit* menu. You can also select *Remove* from the class's popup menu. Both options work for arrows as well: You can either first select the arrow and then select *Remove* from the menu, or you can use the arrow's popup menu.

6 Using the code pad

The BlueJ code pad allows quick and easy evaluation of arbitrary snippets of Java code (expressions and statements). Thus, the code pad can be used to investigate details of Java semantics and to illustrate and experiment with Java syntax.

6.1 Showing the code pad

Summary: To start using the code pad, select Show Code Pad *from the* View *menu.*

The code pad is not shown by default. To show it, use the *Show Code Pad* item from the *View* menu. The main window will now include the code pad interface at the lower right, next to the object bench (Figure 13). Both the horizontal and vertical boundaries of the code pad and object bench can be adjusted to change their sizes.

The code pad area can now be used to enter expressions or statements. On pressing *Enter*, each line will be evaluated and a result may be displayed.

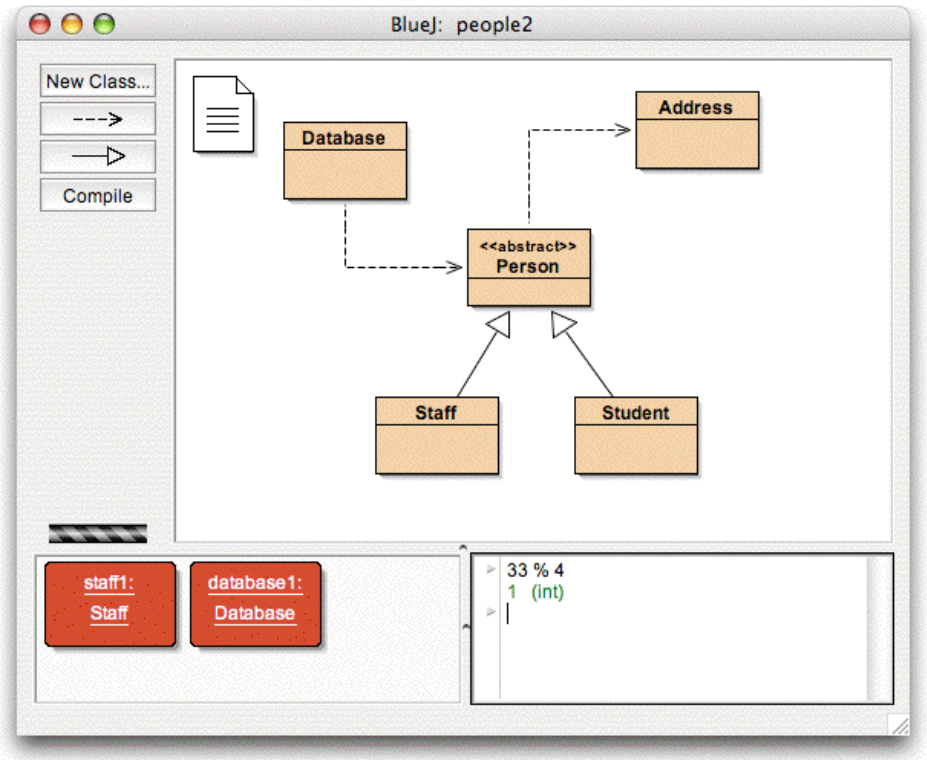

Figure 13: The main window with code pad shown

6.2 Simple expression evaluation

Summary: To evaluate Java expressions, just type them into the code pad.

The code pad can be used to evaluate simple expressions. Try entering, for example:

```
4 + 45
"hello".length()
Math.max(33, 4)
(int) 33.7
javax.swing.JOptionPane.showInputDialog(null, "Name:")
```

Expressions can refer to standard Java values and objects, as well as classes from the current project. The code pad will display the result value, followed by its type (in parenthesis), or an error message if the expression is incorrect.

You can also use the objects you have on the object bench. Try the following: place an object of class student onto the object bench (using the class popup menu as described earlier). Name it *student1*.

In the code pad, you can now type

```
student1.getName()
```

Similarly, you can refer to all available methods from your project classes.

6.3 Receiving objects

Summary: To transfer objects from the code pad to the object bench, drag the small object icon.

Some expression results are objects, rather than simple values. In this case, the result is shown as <*object reference*>, followed by the type of the object, and a small object icon is painted next to the result line (Figure 14).

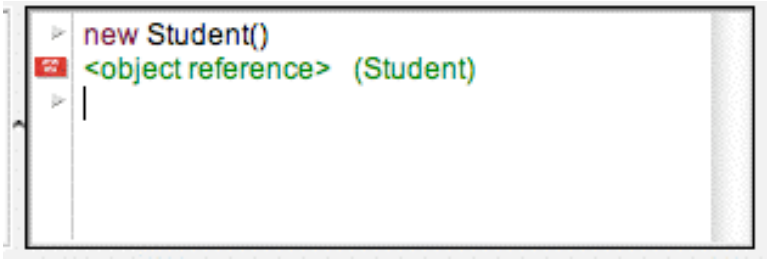

Figure 14: An object as a result of a code pad expression

If the result is a string, the string value will be displayed as the result, but you will also see the small object icon (since strings are objects).

Some expressions you could try to create objects are

```
new Student()
"marmelade".substring(3,8)
new java.util.Random()
"hello" + "world"
```

The small object icon can now be used to continue working with the resulting object. You can point to the icon and drag it onto the object bench (Figure 15). This will place the object onto the bench, where it will be available for further calls to its methods, either via its popup menu or via the code pad.

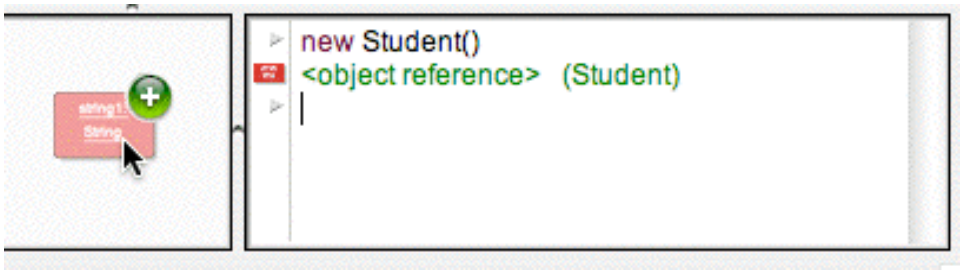

Figure 15: Dragging the object to the object bench

6.4 Inspecting objects

Summary: To inspect result objects in the code pad, double-click the small object icon.

If you want to inspect an object that was returned as a result from a code pad expression, you can do this without placing it onto the object bench: you can just double-click the object's icon to open the usual object inspector.

6.5 Executing statements

Summary: Statements that are typed into the code pad are executed.

You can also use the code pad to execute statements (that is: Java instructions that do not return a value). Try these, for example:

```
System.out.println("Gurkensalat");
System.out.println(new java.util.Random().nextInt(10));
```

Statements are correctly evaluated and executed with or without semicolons at the end.

6.6 Multi-line statements and sequences of statements

Summary: Use shift-Enter *at the end of a line to enter multi-line statements.*

You can enter sequences of statements or statements spanning multiple lines by using *shift-Enter* at the end of the input line. Using *shift-enter* will move the cursor to the start of the next line, but not (yet) execute the input. At the end of the last input line type *Enter* to evaluate all lines together. Try, for example, a *for* loop:

```
for (int i=0; i<5; i++) {
   System.out.println("number: " + i);
}
```

6.7 Working with variables

Summary: Local variables can be used in single, multi-line statements. The names of objects on the object bench serve as instance fields.

Variables – instance fields and local variables – can be used in the code pad in restricted ways.

You can declare local variables in the code pad, but this is only useful as part of multi-line statement sequences, since variables are discarded between separate inputs. For example: You can enter the following block as a single, multi-line input, and it will work as expected:

```
int sum;
sum = 0;
for (int i=0; i<100; i++) {
   sum += i;
}
System.out.println("The sum is: " + sum);
```

Entering the same sequence as separate input statements, however, will fail, since the local variable *sum* is not remembered between inputs.

You can think of the input you type as the text within a method body. All code that is legal to write in the body of a Java method is also legal in the code pad. However, every text input you type forms the part of a *different* method, so you cannot refer from one input line to a variable declared in another.

You can think of objects on the object bench as instance fields. You cannot define any new instance fields from within a method body (or from within the code pad), but you can refer to the instance fields and make calls to the objects held in them.

You can create a new instance field by dragging an object from the code pad to the object bench.

6.8 Command history

Summary: Use up-arrow *and* down-arrow *keys to make use of the input history.*

The code pad keeps a history of your previously used inputs. Using the *up* or *down* arrow keys, you can easily recall previous input lines, which can be edited before being reused.

7 Debugging

This section introduces the most important aspects of the debugging functionality in BlueJ. In talking to computing teachers, we have very often heard the comment that using a debugger in first year teaching would be nice, but there is just no time to introduce it. Students struggle with the editor, compiler and execution; there is no time left to introduce another complicated tool.

That's why we have decided to make the debugger as simple as possible. The goal is to have a debugger that you can explain in 15 minutes, and that students can just use from then on without further instruction. Let's see whether we have succeeded.

First of all, we have reduced the functionality of traditional debuggers to three tasks:

- setting breakpoints
- stepping through the code
- inspecting variables

In return, each of the three tasks is very simple. We will now try out each one of them.

To get started, open the project *debugdemo*, which is included in the *examples* directory in the distribution. This project contains a few classes for the sole purpose of demonstrating the debugger functionality – they don't make a lot of sense otherwise.

7.1 Setting breakpoints

Summary: To set a breakpoint, click in the breakpoint area to the left of the text in the editor.

Setting a breakpoint lets you interrupt the execution at a certain point in the code. When the execution is interrupted, you can investigate the state of your objects. It often helps you to understand what is happening in your code.

In the editor, to the left of the text, is the breakpoint area (Figure 16). You can set a breakpoint by clicking into it. A small stop sign appears to mark the breakpoint. Try this now. Open the class *Demo*, find the method *loop*, and set a breakpoint somewhere in the *for* loop. The stop sign should appear in your editor.

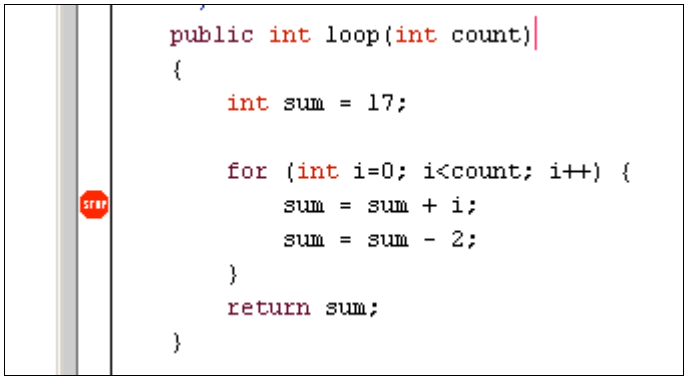

Figure 16: A breakpoint

When the line of code is reached that has the breakpoint attached, execution will be interrupted. Let's try that now.

Create an object of class *Demo* and call the *loop* method with a parameter of, say, 10. As soon as the breakpoint is reached, the editor window pops up, showing the current line of code, and a debugger window pops up. It looks something like Figure 17.

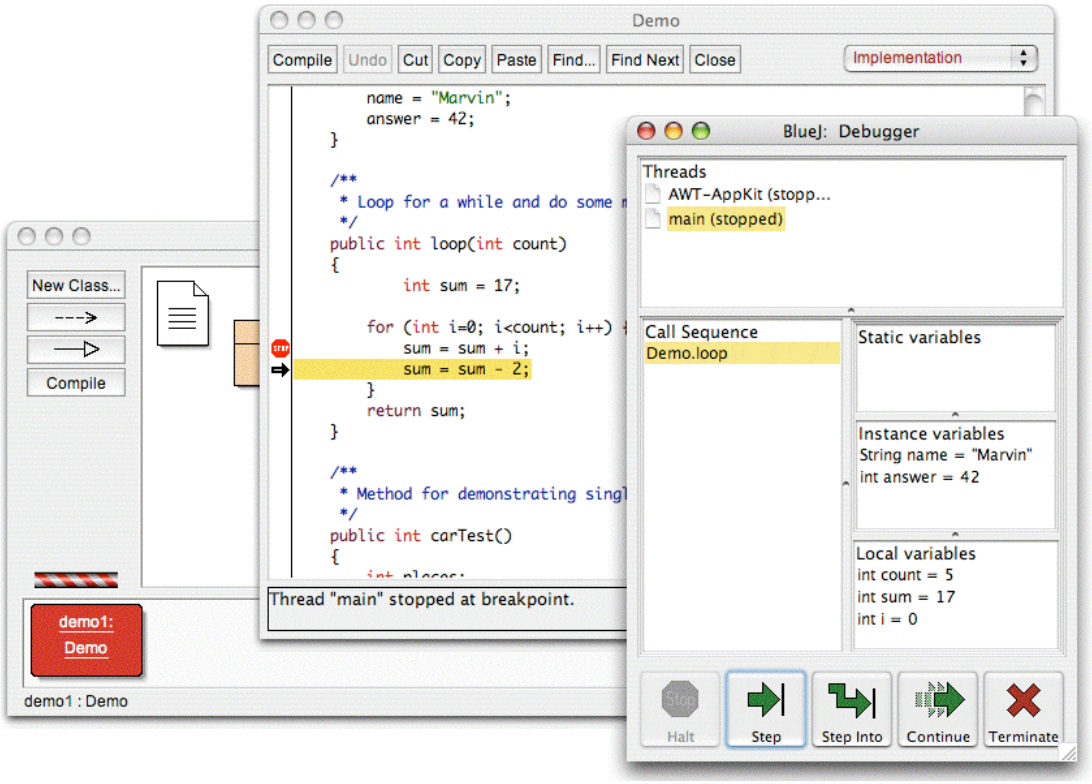

Figure 17: The debugger window

The highlight in the editor shows the line that will be executed next. (The execution is stopped *before* this line was executed.)

7.2 Stepping through the code

Summary: To single-step through your code, use the Step *and* Step Into *buttons in the debugger.*

Now that we have stopped the execution (which convinces us that the method really does get executed and this point in the code really does get reached), we can single-step through the code and see how the execution progresses. To do this, repeatedly click on the *Step* button in the debugger window. You should see the source line in the editor changing (the highlight moves with the line being executed). Every time you click the *Step* button, one single line of code gets executed and the execution stops again. Note also that the values of the variables displayed in the debugger window change (for example the value of *sum*.) So you can execute step by step and observe what happens. Once you get tired of this, you can click on the breakpoint again to remove it, and then click the *Continue* button in the debugger to restart the execution and continue normally.

Let's try that again with another method. Set a breakpoint in class *Demo*, method *carTest()*, in the line reading

```
places = myCar.seats();
```

Call the method. When the breakpoint is hit, you are just about to execute a line that contains a method call to the method *seats()* in class *Car*. Clicking *Step* would step over the whole line. Let's try *Step Into* this time. If you *step into* a method call, then you enter the method and execute that method itself line by line (not as a single step). In this case, you are taken into the *seats()* method in class *Car*. You can now happily step through this method until you reach the end and return to the calling method. Note how the debugger display changes.

Step and *Step Into* behave identically if the current line does not contain a method call.

7.3 Inspecting variables

Summary: Inspecting variables is easy – they are automatically displayed in the debugger.

When you debug your code, it is important to be able to inspect the state of your objects (local variables and instance variables).

Doing it is trivial – most of it you have seen already. You do not need special commands to inspect variables; static variables, instance variables of the current object and local variables of the current method are always automatically displayed and updated.

You can select methods in the call sequence to view variables of other currently active objects and methods. Try, for example, a breakpoint in the *carTest()* method

again. On the left side of the debugger window, you see the call sequence. It currently shows

```
Car.seats
Demo.carTest
```

This indicates that `Car.seats` was called by `Demo.carTest`. You can select `Demo.carTest` in this list to inspect the source and the current variable values in this method.

If you step past the line that contains the `new Car(...)` instruction, you can observe that the value of the local variable `myCar` is shown as <*object reference*>. All values of object types (except for Strings) are shown in this way. You can inspect this variable by double-clicking on it. Doing so will open an object inspection window identical to those described earlier (section 4.1). There is no real difference between inspecting objects here and inspecting objects on the object bench.

7.4 Halt and terminate

Summary: Halt *and* Terminate *can be used to halt an execution temporarily or permanently.*

Sometimes a program is running for a long time, and you wonder whether everything is okay. Maybe there is an infinite loop, maybe is just takes this long. Well, we can check. Call the method *longloop()* from the *Demo* class. This one runs a while.

Now we want to know what's going on. Show the debugger window, if it is not already on screen.

Now click the *Halt* button. The execution is interrupted just as if we had hit a breakpoint. You can now step a couple of steps, observe the variables, and see that this is all okay – it just needs a bit more time to complete. You can just *Continue* and *Halt* several times to see how fast it is counting. If you don't want to go on (for example, you have discovered that you really are in an infinite loop) you can just hit *Terminate* to terminate the whole execution. Terminate should not be used too frequently – you can leave perfectly well written objects in an inconsistent state by terminating the machine, so it is advisable to use it only as an emergency mechanism.

8 Creating stand-alone applications

Summary: To create a stand-alone application, use Project - Create Jar File...

BlueJ can create executable jar files. Executable jar files can be executed on some systems by double-clicking the file (for example on Windows and MacOS X), or by issuing the command `java -jar <file-name>.jar` (Unix or DOS prompt).

We will try this with the example project *hello*. Open it (it is in the *examples* directory). Make sure that the project is compiled. Select the *Create Jar File...* function from the *Project* menu.

A dialogue opens that lets you specify the main class (Figure 18). This class must have a valid *main method* defined (with the signature `public static void main(String[] args)`).

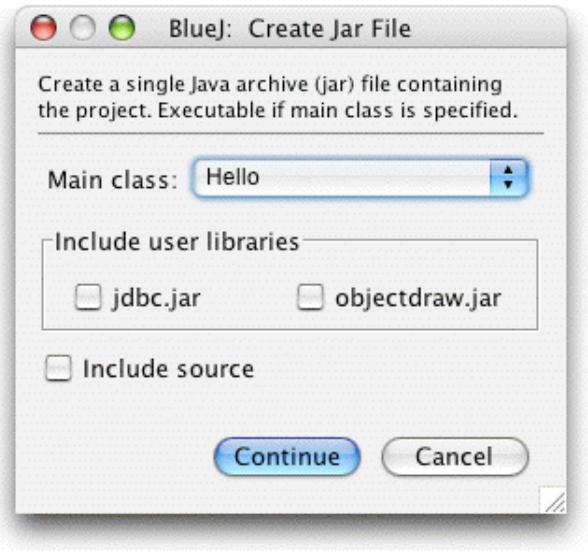

Figure 18: The "Create Jar File" dialogue

In our example, choosing the main class is easy: there is only one class. Select *Hello* from the popup menu. If you have other projects, select the class that holds the "main" method you want to execute.

Usually, you would not include sources in executable files. But you can, if you want to distribute your sources as well. (You can use the jar format to send your whole project to someone else via email in a single file, for example.)

If you have configured BlueJ to use user libraries (either via the *Preferences/Libraries* setting, or using the *lib/userlib* directory) you will see an area titled *Include user libraries* in the middle of the dialogue. (If you are not using any libraries, this area will be absent.) You should check every library that your current project uses.

Click *Continue*. Next, you see a file chooser dialogue that lets you specify a name for the jar file to create. Type *hello* and click Create.

If you do not have libraries to be included, a file *hello.jar* will now be created. If you have libraries, a directory named *hello* will be created, and within it the jar file *hello.jar*. The directory also contains all necessary libraries. Your jar file expects to find referenced libraries in the same directory it is in itself – so make sure to keep these jar files together when you move them around.

You can double-click the jar file only if the application uses a GUI interface. Our example uses text I/O, so we have to start it from a text terminal. Let's try to run the jar file now.

Open a terminal or DOS window. Then go to the directory where you saved your jar file (you should see a file *hello.jar*). Assuming Java is installed correctly on your system, you should then be able to type

```
java -jar hello.jar
```

to execute the file.

9 Creating applets

9.1 Running an applet

Summary: To run an applet, select Run Applet *from the applet's popup menu.*

BlueJ allows creating and executing applets as well as applications. We have included an applet in the examples directory in the distribution. First, we want to try executing an applet. Open the *appletdemo* project from the examples.

You will see that this project has only one class; it is named *CaseConverter*. The class icon is marked (with the tag <<applet>>) as an applet. After compiling, select the *Run Applet* command from the class's popup menu.

A dialogue pops up that lets you make some selections (Figure 19).

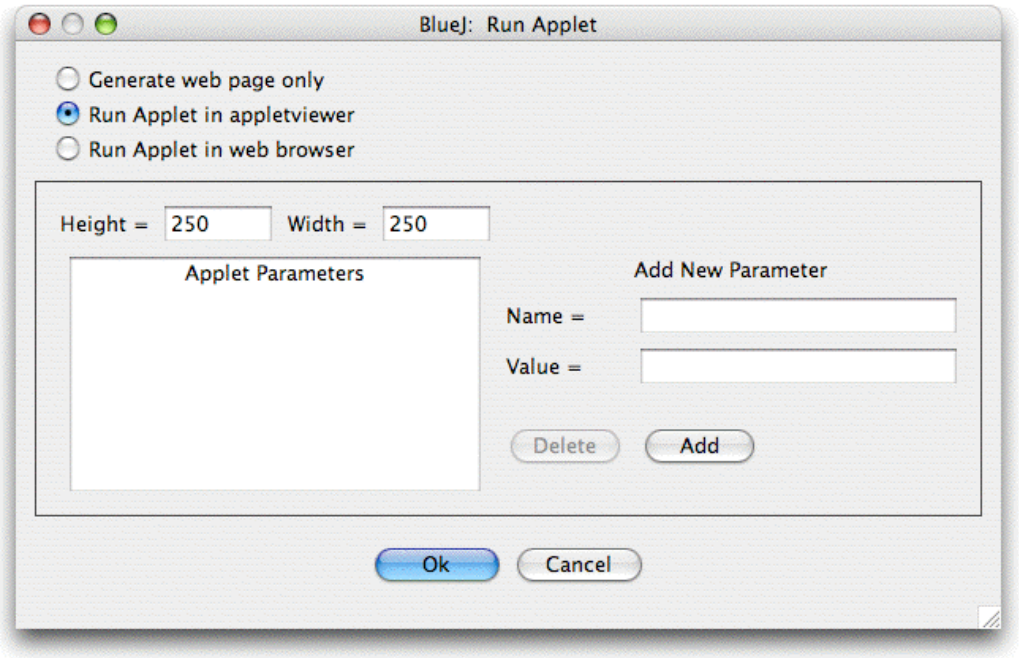

Figure 19: The "Run Applet" dialogue

You see that you have a choice of running the applet in a browser or in an applet viewer (or just to generate the web page without running it). Leave the default settings and click *OK*. After a few seconds, an applet viewer should pop up displaying the case converter applet.

The applet viewer is installed together with your J2SE SDK (your Java installation), so it is always guaranteed to be of the same version as your Java compiler. It

generally causes fewer problems than browsers do. Your web browser may run a different version of Java and, depending on which version of which browser you use, may cause problems. With most current browsers it should work fine, though.

On Microsoft Windows and MacOS systems, BlueJ uses your default browser. On Unix systems, the browser is defined in the BlueJ settings.

9.2 Creating an applet

Summary: To create an applet, click the New Class *button and select* Applet *as the class type.*

After having seen how to run an applet, we want to create our own.

Create a new class with *Applet* as the class type (you can select the type in the *New Class* dialogue). Compile, then run the applet. That's it! That wasn't too bad, was it?

Applets (like other classes) are generated with a default class skeleton that contains some valid code. For applets, this code shows a simple applet with two lines of text. You can now open the editor and edit the applet to insert your own code.

You will see that all the common applet methods are there, each with a comment explaining its purpose. The sample code is all in the *paint* method.

9.3 Testing the applet

In some situations it can be useful to create an applet object on the object bench (as for normal classes). You can do that – the constructor is shown in the applet's popup menu. From the object bench you cannot execute the full applet, but you can call some methods. This may be useful to test single methods you may have written as part of your applet implementation.

If you set breakpoints in an applet, they will have no effect when you run the applet in an applet viewer or a web browser. This is because both the applet viewer and web browsers use their own virtual machines to execute the applet, which do not understand anything about BlueJ breakpoints.

If you want to use breakpoints and single-stepping in an applet, you can use the class *AppletWindow*, written by Michael Trigoboff. The class provides a frame that lets you run the applet directly within BlueJ, so normal debugging methods work. You can find this class and a demo in the *Resources* section of the BlueJ web site.

10 Other operations

10.1 Opening non-BlueJ packages in BlueJ

Summary: Non-BlueJ packages can be opened with the Project: Open Non BlueJ... *command.*

BlueJ lets you open existing packages that were created outside of BlueJ. To do this, select Project – Open Non BlueJ... from the menu. Select the directory that contains the Java source files, then click the Open in BlueJ button. The system will ask for confirmation that you want to open this directory.

10.2 Adding existing classes to your project

Summary: Classes can be copied into a project from outside by using the Add Class from File... *command.*

Often, you want to use a class that you got from somewhere else in your BlueJ project. For example, a teacher may give a Java class to students to be used in a project. You can easily incorporate an existing class into your project by selecting Edit – Add Class from File... from the menu. This will let you select a Java source file (with a name ending in *.java*) to be imported.

When the class is imported into the project, a copy is taken and stored in the current project directory. The effect is exactly the same as if you had just created that class and written all its source code.

An alternative is to add the source file of the new class to the project directory from outside BlueJ. Next time you open that project, the class will be included in the project diagram.

10.3 Calling *main* and other static methods

Summary: Static methods can be called from the class's popup menu.

Open the *hello* project from the *examples* directory. The only class in the project (class *Hello*) defines a standard main method.

Right-click on the class, and you will see that the class menu includes not only the class's constructor, but also the static *main* method. You can now call *main* directly from this menu (without first creating an object).

All static methods can be called like this. The standard main method expects an array of Strings as an argument. You can pass a String array using the standard Java syntax for array constants. For example, you could pass

```
{"one", "two", "three"}
```

(including the braces) to the method. Try it out!

> Side note: *In standard Java, array constants cannot be used as actual arguments to method calls. They can only be used as initialisers. In BlueJ, to enable interactive calls of standard main methods, we allow passing of array constants as parameters.*

10.4 Generating documentation

Summary: To generate documentation for a project, select Project Documentation *from the* Tools *menu.*

You can generate documentation for your project in the standard *javadoc* format from within BlueJ. To do this, select the Tools - Project Documentation from the menu. This function will generate the documentation for all classes in a project from the classes' source code and open a web browser to display it.

You can also generate and view the documentation for a single class directly in the BlueJ editor. To do this, open the editor and use the popup menu in the editor's toolbar. Change the selection from *Implementation* to *Interface*. This will show the *javadoc* style documentation (the class's interface) in the editor.

10.5 Working with libraries

Summary: The Java standard class API can be viewed by selecting Help - Java Class Libraries.

Frequently, when you write a Java program, you have to refer to the Java standard libraries. You can open a web browser showing the JDK API documentation by selecting Help - Java Standard Classes from the menu (if you are online).

The JDK documentation can also be installed and used locally (offline). Details are explained in the help section on the BlueJ web site.

10.6 Creating objects from library classes

Summary: To create objects from library classes, use Tools – Use Library Class.

BlueJ also offers a function to create objects from classes that are not part of your project, but defined in a library. You can, for example, create objects of class String or ArrayList. This can be very useful for quick experimentation with these library objects.

You can create a library object by selecting Tools – Use Library Class... from the menu. A dialog will pop up that prompts you to enter a fully qualifies class name, such as *java.lang.String*. (Note that you must type the fully qualified name, that is the name including the package names that contain the class.)

The text entry field has an associated popup menu showing recently used classes. Once a class name has been entered, pressing *Enter* will display all constructors and static methods of that class in a list in the dialog. Any of these constructors or static methods can now be invoked by selecting them from this list.

The invocation proceeds as any other constructor or method call.

11 Just the summaries

Getting started

1. To open a project, select *Open* from the *Project* menu.
2. To create an object, select a constructor from the class popup menu.
3. To execute a method, select it from the object popup menu.
4. To edit the source of a class, double-click its class icon.
5. To compile a class, click the *Compile* button in the editor. To compile a project, click the *Compile* button in the project window.
6. To get help for a compiler error message, click the question mark next to the error message.

Doing a bit more...

7. Object inspection allows some simple debugging by checking an object's internal state.
8. An object can be passed as a parameter to a method call by clicking on the object icon.

Creating a new project

9. To create a project, select *New...* from the *Project* menu.
10. To create a class, click the *New Class* button and specify the class name.
11. To create an arrow, click the arrow button and drag the arrow in the diagram, or just write the source code in the editor.
12. To remove a class or an arrow, select the remove function from its popup.

Using the code pad

13. To start using the code pad, select Show Code Pad from the View menu.
14. To evaluate Java expressions, just type them into the code pad.
15. To transfer objects from the code pad to the object bench, drag the small object icon.
16. To inspect result objects in the code pad, double-click the small object icon.
17. Statements that are typed into the code pad are executed.
18. Use shift-Enter at the end of a line to enter multi-line statements.
19. Local variables can be used in single, multi-line statements. The names of objects on the object bench serve as instance fields.
20. Use up-arrow and down-arrow keys to make use of the input history.

Debugging

21. To set a breakpoint, click in the breakpoint area to the left of the text in the editor.
22. To single-step through your code, use the *Step* and *Step Into* buttons in the debugger.
23. Inspecting variables is easy – they are automatically displayed in the debugger.
24. *Halt* and *Terminate* can be used to halt an execution temporarily or permanently.

Creating stand-alone applications

25. To create a stand-alone application, use *Project - Create Jar File...*

Creating applets

26. To run an applet, select *Run Applet* from the applet's popup menu.
27. To create an applet, click the *New Class* button and select *Applet* as the class type.

Other operations

28. Non-BlueJ packages can be opened with the *Project: Open Non BlueJ...* command.
29. Classes can be copied into a project from outside by using the *Add Class from File...* command.
30. Static methods can be called from the class's popup menu.
31. To generate documentation for a project, select *Project Documentation* from the *Tools* menu.
32. The Java standard class API can be viewed by selecting *Help - Java Standard Libraries*.
33. To create objects from library classes, use *Tools – Use Library Class.*