The BlueJ Tutorial

Version 1.0

for BlueJ Version 1.0

Michael Kölling
School of Network Computing
Monash University

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 Monash University, Melbourne, Australia.

More information about BlueJ is available at http://bluej.monash.edu.

1.2 Scope and audience

This tutorial is aimed at people wanting to familiarise 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.

It is assumed that the reader is familiar with the Java programming language – no attempt is made in this tutorial to introduce Java.

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.

Most sections end with a one-line end-of-section summary sentence. 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@monash.edu.au).

2 Getting started

2.1 Installation

BlueJ is distributed as an archive of Java classes in "jar" format. Installing it is quite straightforward.

Prerequisites

You must have JDK 1.2.2 or later installed on your system to use BlueJ. If you do not have JDK installed you can download it from Sun's web site at http://www.javasoft.com/products/jdk.

Getting BlueJ

The BlueJ distribution file is named *bluej*-xxx.*jar*, where *xxx* is a version number. For example, the BlueJ version 1.0.2 distribution is named *bluej-102.jar*. You might get this file on disk, or you can download it from the BlueJ web site at http://bluej.monash.edu.

About SDK, JDK and JRE

There sometimes is some confusion about different Java distributions: SDK, JDK and JRE packages. You should install the latest version of the *Java 2 SDK* (Software Development Kit). The term JDK (Java Development Kit) is an older name for the same thing. Sun have changed their naming convention at some stage, but sometimes the older name (JDK) is still used. For example, if you install *Java 2 SDK v. 1.3*, then the default installation directory is called *jdk1.3*.

The JRE (Java Runtime Environment) is different: It is a subset of the SDK for Java execution. For BlueJ that is not enough. We need the SDK because it includes some development tools that BlueJ uses. JRE automatically gets installed as part of the SDK installation. On Windows systems, the JRE version is the default – if you do not use a full path to specify the JDK version, Java executes with the JRE system. This is a problem – you can avoid it by using the full path (by default `c:\jdk1.3\bin\java`) to start Java.

Installing

Create a directory named *bluej* and move the distribution file into it. Go to a command prompt (in Windows: a DOS prompt) and execute the following command. NOTE: For this example, I use the distribution file *bluej-103.jar* – you need to use the file name of the file you've got (with the correct version number).

Unix:

```
<jdk-path>/bin/java -jar bluej-103.jar
```

Windows:

```
<jdk-path>\bin\java -jar bluej-103.jar
```

<jdk-path> is the directory, where JDK was installed. For example, using the default installation for JDK 1.3 on Windows, the command would be

```
C:\jdk1.3\bin\java -jar bluej-103.jar
```

A window pops up, letting you enter some details. Click *Install*. After finishing, BlueJ should be installed.

If you have any problems, check the FAQ on the BlueJ web site.

2.2 Starting BlueJ

The BlueJ installation installs a script named *bluej* in the installation directory. From a GUI interface, just double-click the file. From a command line (e.g. Unix or DOS), you can start BlueJ with or without a package as an argument:

```
$ bluej
```

or

```
$ bluej examples/people
```

2.3 Opening a package

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

If you start BlueJ from a command line, and you give a package as an argument, it will automatically be opened. If you start BlueJ without an argument, use the Package – Open... menu command to select and open a package.

3 The basics – edit / compile / execute

For this tutorial section, open the project *people*, which is included in the BlueJ distribution. You can find it in the *examples* directory in the BlueJ home directory. After opening the package 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.

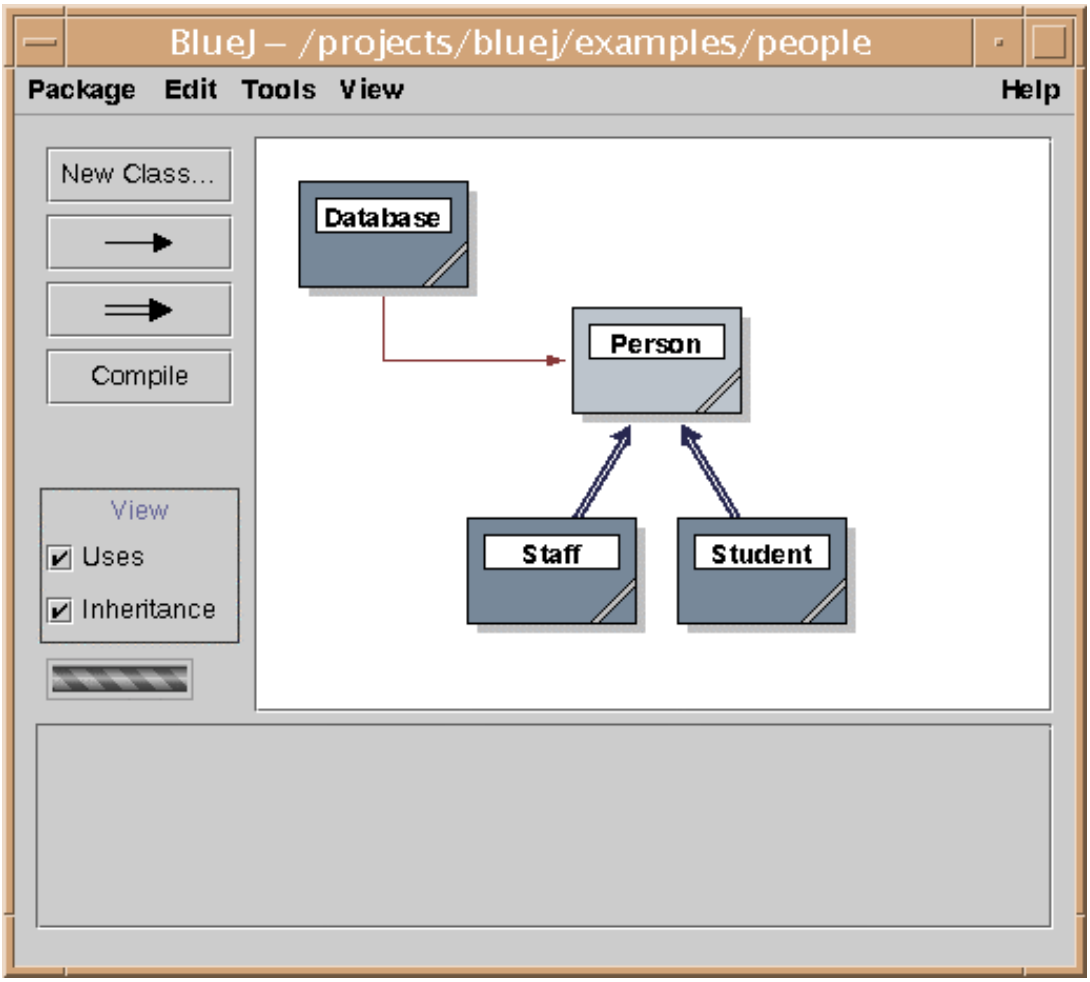

Figure 1: The BlueJ main window

3.1 Creating objects

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 (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.

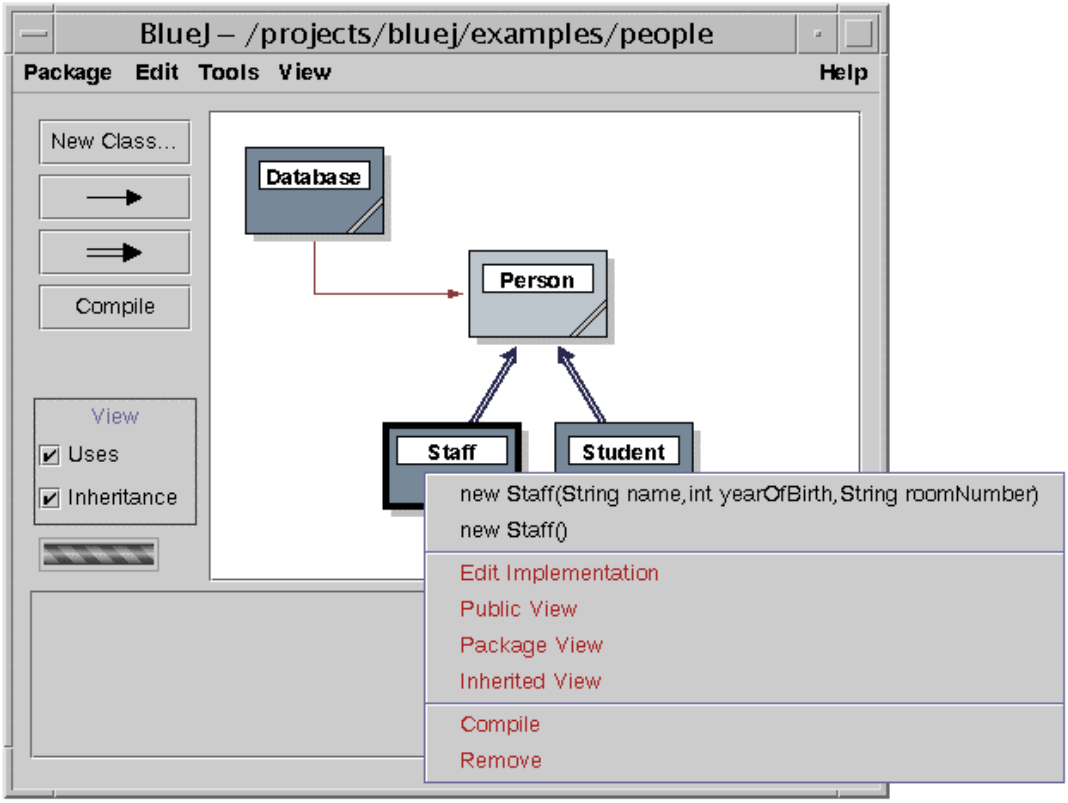

Figure 2: Class operations (popup menu)

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.

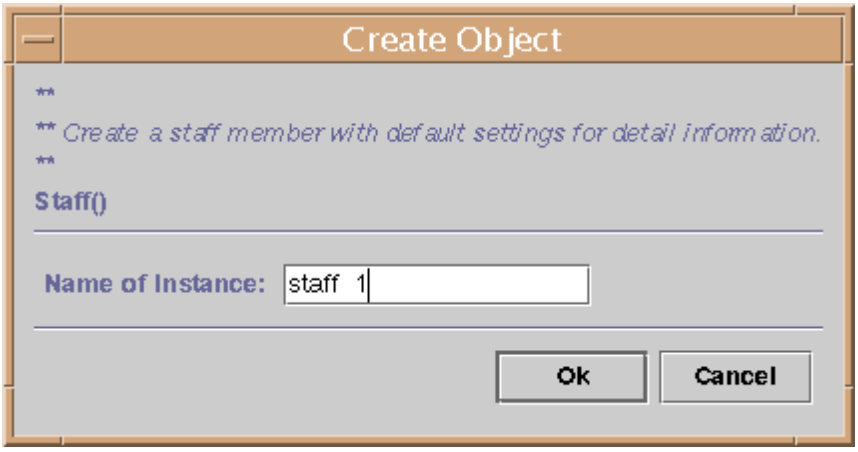

Figure 3: Object creation without parameters

This dialogue asks you for a name for the object to be created. At the same time, a default name (*staff_1*) 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.

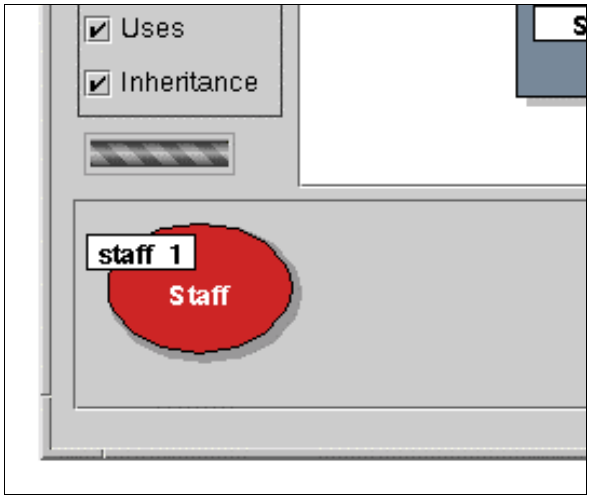

Figure 4: An object on the object bench

You might have noticed that the class *Person* has a different colour than the other classes. This colour identifies an abstract class. You will notice (if you try) that you cannot create objects of abstract classes (as the Java language specification defines).

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

3.2 Execution

Now that you've created an object, you can execute its public operations. 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's concentrate on the methods.

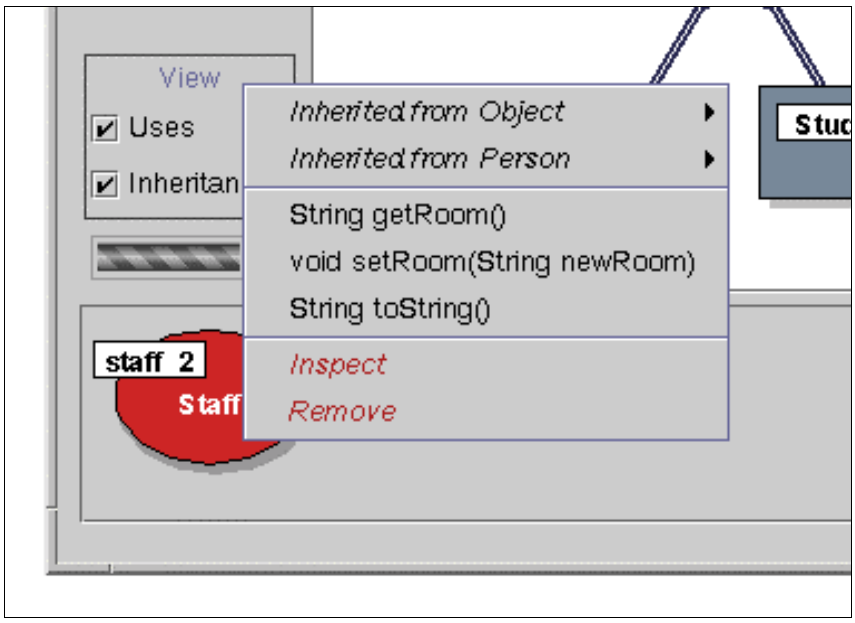

Figure 5: The object menu

You see that there are methods *getRoom* and *setRoom* 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.

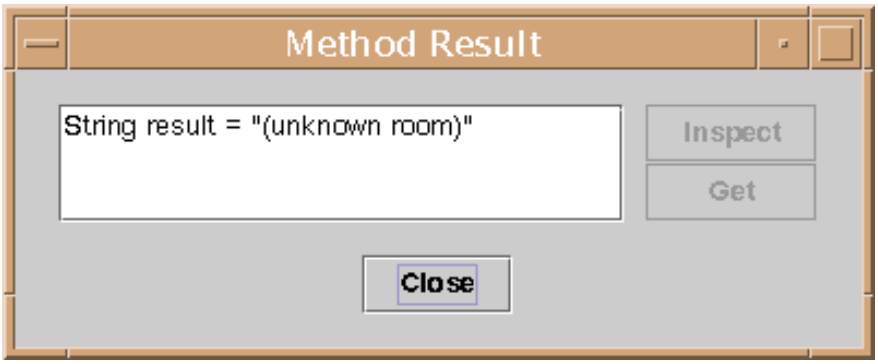

Figure 6: Display of a function result

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 6). 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's try to specify a room name. 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 the parameters (Figure 7).

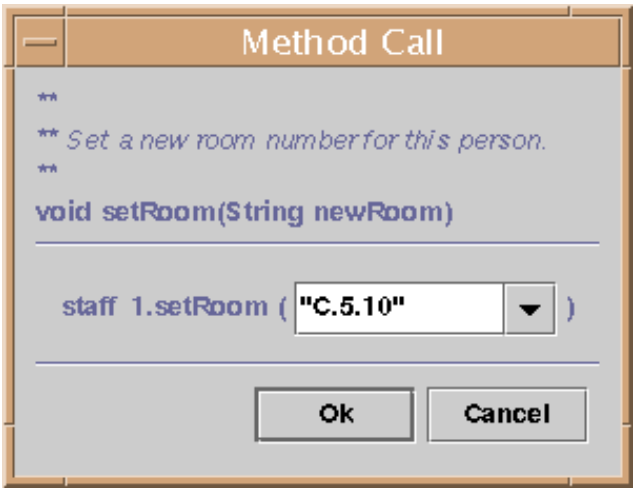

Figure 7: Function call dialogue with parameters

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 parameters. The signature at the top tells us that one parameter of type String is expected. Enter the new name 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 *getName* again to check that the name 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.

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

3.3 Editing a class

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 *Edit Implementation* 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's change the method by adding the prefix "`room`" to the function result (so that the method returns, say, "room C.5.10" instead of just "C.5.10"). 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.

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

3.4 Compilation

After inserting the text (before you do anything else), check the package 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 package. This is because the classes in the people package were distributed already compiled. Often BlueJ packages are distributed uncompiled, so expect to see most class icons striped when you first open a package 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 package window also has a *Compile* button. This compile operation compiles the whole package. (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.

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

3.5 Help with compiler errors

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).

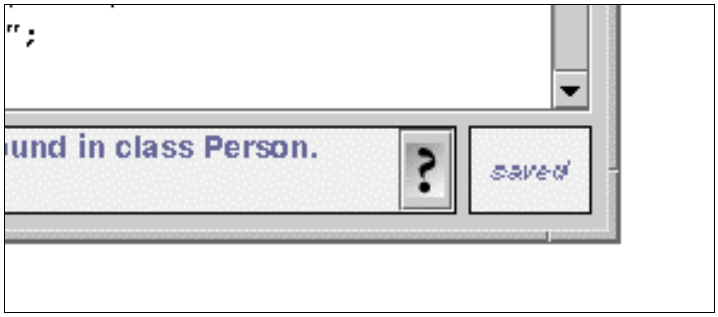

Figure 8: A compiler error and the *Help* button

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.

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

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

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).

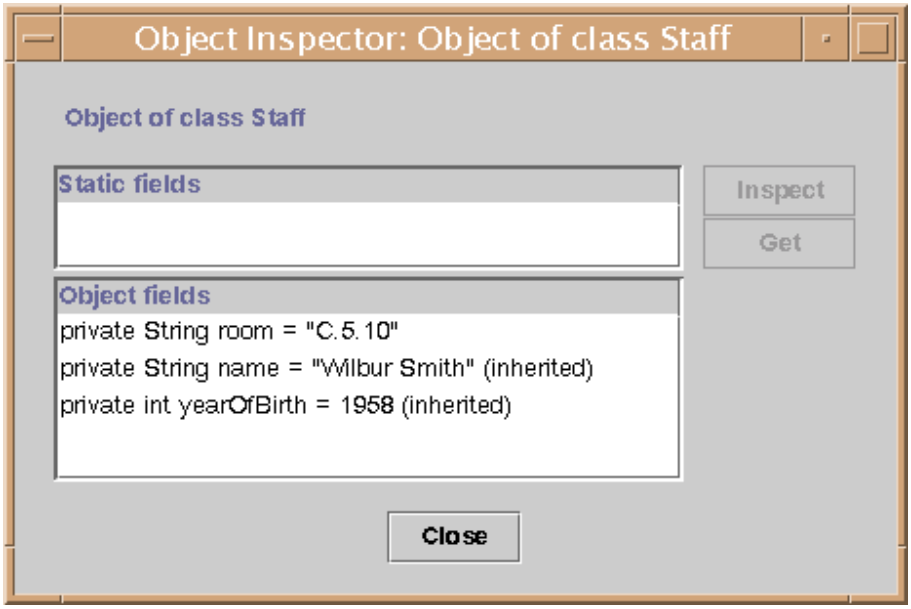

Figure 9: Inspection dialogue

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 package. Open the package

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*.

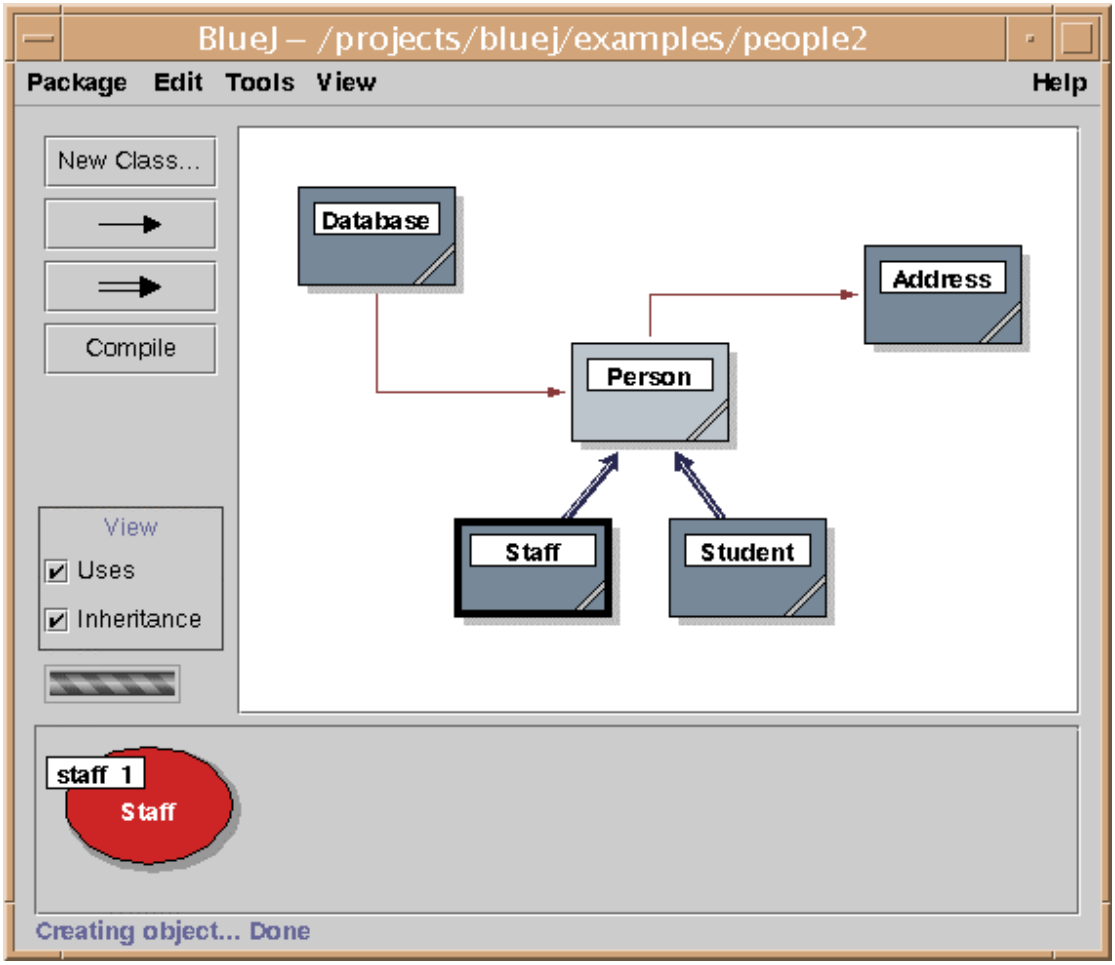

Figure 10: The *people2* package window

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 <*object reference*> – 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).

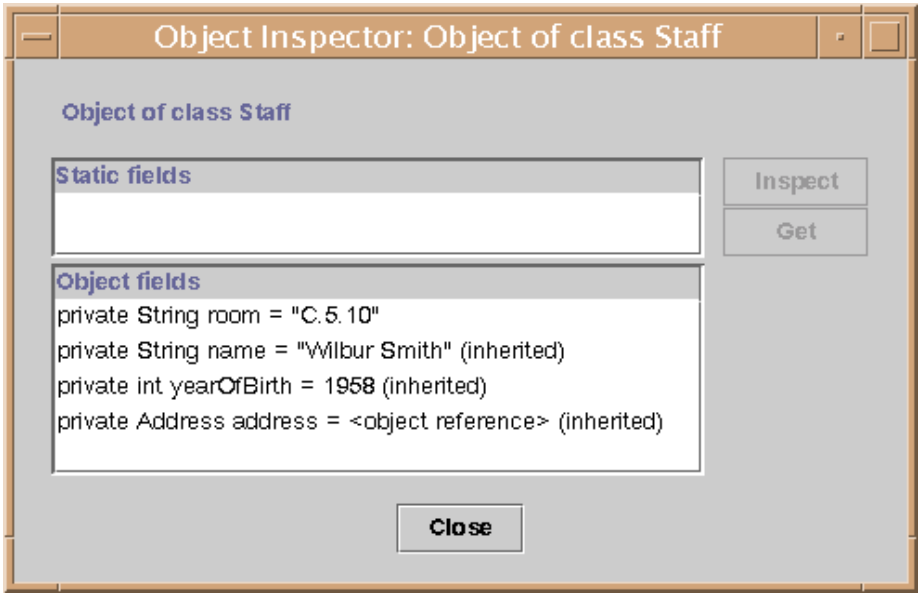

Figure 11: Inspection with object reference

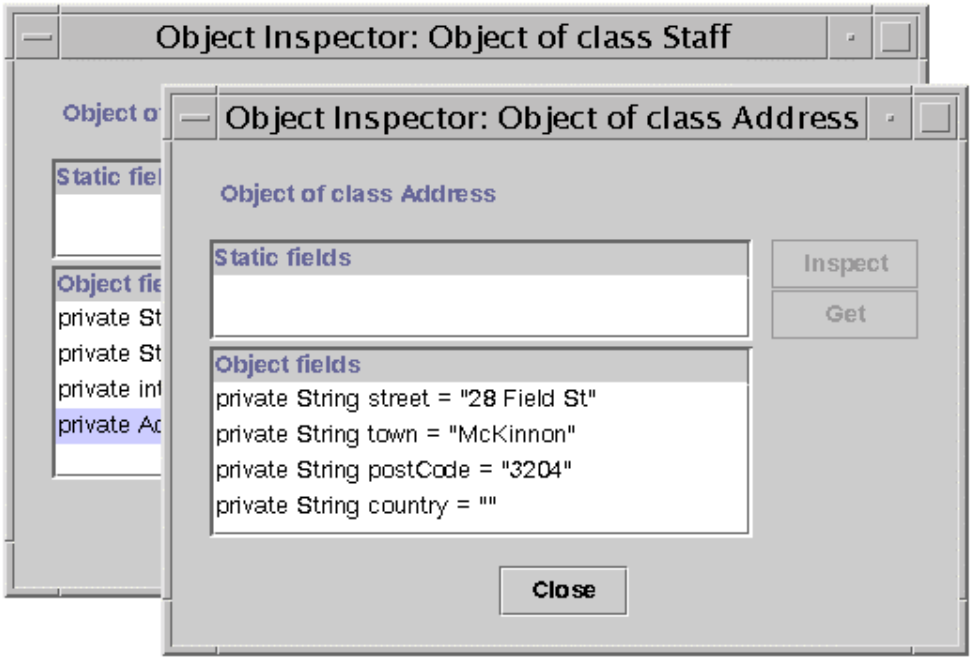

Figure 12: Inspection of internal object

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.

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

4.2 Composition

The term “composition” refers to the ability to pass objects as parameters to other objects. Let’s 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”.

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

5 Creating a new package

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

5.1 Creating the package directory

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

Summary: To create a package, select New... from the Package menu.

5.2 Creating classes

You can now create your classes by clicking the *New Class* button on the package 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. Different colours identify the different types of classes, for example blue for normal classes, lighter blue for abstract classes, green for interfaces. 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.

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

5.3 Creating dependencies

The class diagram shows dependencies between classes in the form of arrows. Inheritance relations ("extends" or "implements") are shown as double arrows; "uses" relations are shown as single arrows.

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 (double arrow for "extends" or "implements", single 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.

> Side note: *In BlueJ version 1.0, classes from other packages are not actually displayed in the class diagram. In a future version, they will be displayed as well.*

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.)

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.

5.4 Removing elements

To remove a class from the diagram, select the class and then select *Remove Class* from the *Edit* menu. You can also select *Remove* from the class's popup menu. To remove an arrow, select *Remove Arrow* from the menu and then select the arrow you want to remove.

Summary: To remove a class, select the remove function from its popup menu. To remove an arrow, select remove from the Edit menu and click on the arrow.

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

6.1 Setting breakpoints

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 13). 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.

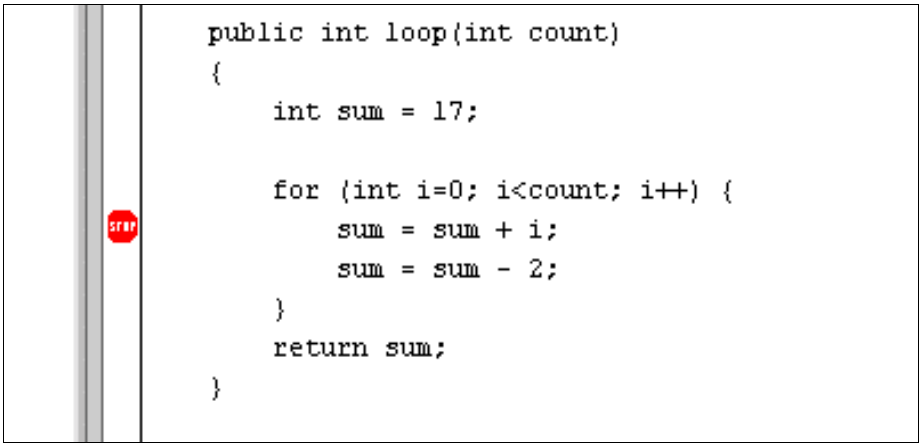

Figure 13: A breakpoint

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 14.

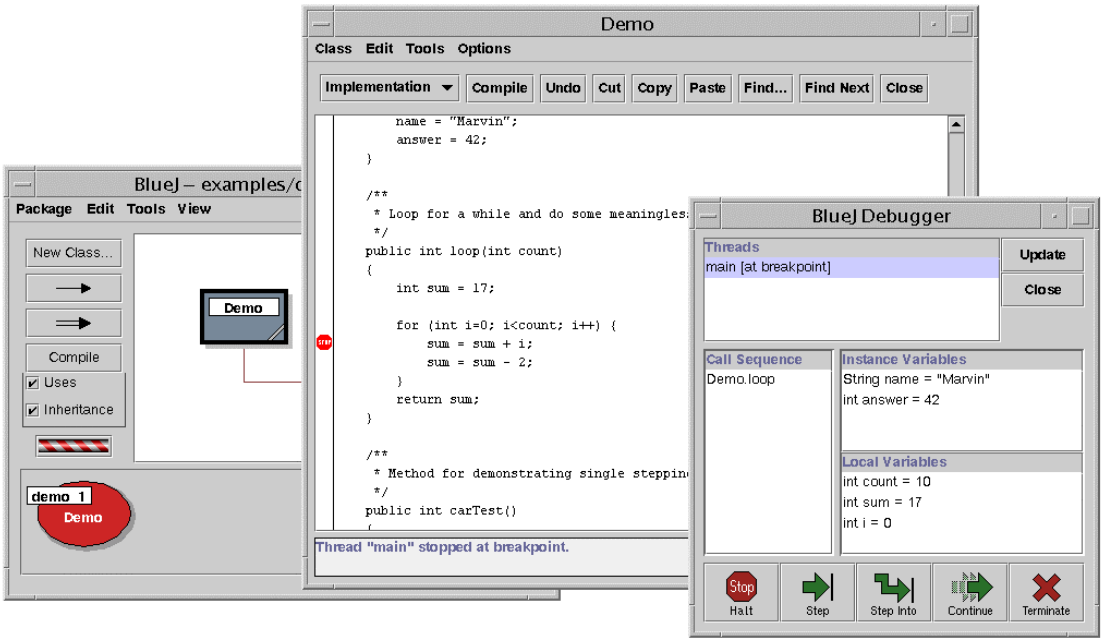

Figure 14: The debugger window

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

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

6.2 Stepping through the code

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.

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

6.3 Inspecting variables

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; 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.

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

6.4 Halt and terminate

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. (By the way, clicking the turning symbol that indicates that the machine is running during execution is a shortcut to showing the debugger.)

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.

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

7 Working with libraries

The library browser is not included in BlueJ version 1.0. It will be released in a later version.

For the moment, libraries are treated as in every other Java environment: look up the library in the JDK documentation and write the appropriate import statement into your source file. You can open a web browser showing the JDK 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 BlueJ reference manual.

8 Creating stand-alone applications

The current version of BlueJ does not yet implement the support for creating applications for distribution automatically. It is easy enough to do manually, though.

BlueJ projects are stored in a directory. This directory contains the usual Java files (source files and class files, nested directories for nested packages) in standard format. It also contains some BlueJ specific files: a file named `bluej.pkg`, a file named `bluej.pkh` and several files ending in `.ctxt`. To create a standard Java application, simply remove these BlueJ-specific files. (If you later want to open it in BlueJ again, you can do so – BlueJ can open standard Java applications. You will, however, lose the layout of your classes on screen.)

9 Creating applets

9.1 Running an applet

BlueJ allows creating and executing applets as well as applications. We have included some applets in the examples directory in the distribution. First, we want to try running one of those. Open the *appletClock* package from the examples.

You will see that this package has only one class; it is named *Clock*. The class icon is marked (with the letters WWW) as an applet. Select the *Run Applet* command from the class's popup menu.

A dialogue pops up that lets you make some selections (Figure 15).

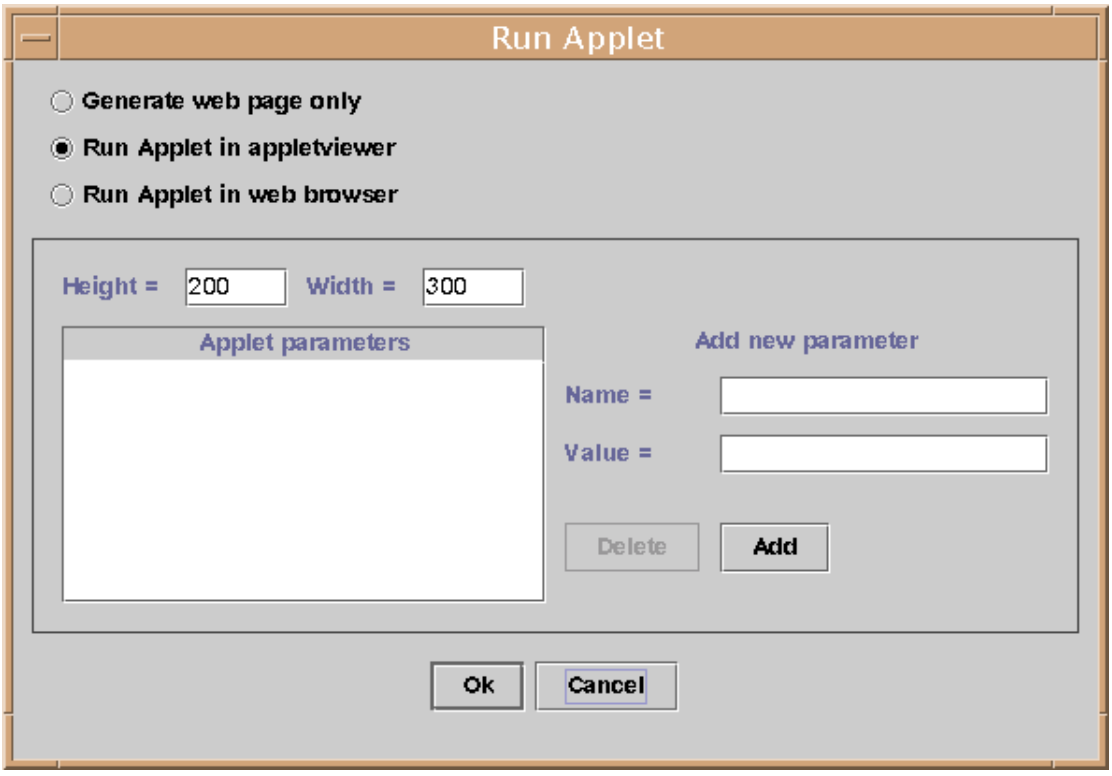

Figure 15: The "Run Applet" dialogue

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 clock applet.

The applet viewer is installed together with your JDK, 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 systems, BlueJ uses your default browser. On Unix systems, the browser is defined in the BlueJ settings.

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

9.2 Creating an applet

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.

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

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.

10 Other Operations

10.1 Opening non-BlueJ packages in BlueJ

BlueJ lets you open existing packages that were created outside of BlueJ. To do this, select Package – Open... 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.

At the time of writing this (BlueJ version 1.0.3), only flat packages which do not contain sub-packages can be opened. This will be improved in a later version.

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

10.2 Importing classes into your package

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 Package – Import Class 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.

Summary: Classes can be copied into a package from outside by using the Import Class *command.*

10.3 Calling *main* and other static methods

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, as we would expect for a static method).

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.*

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

11 Just the summaries

Here is a list of the end-of-section summary lines at a glance.

The basics

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

Doing a bit more...

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

Creating a new package

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

Debugging

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

Creating applets

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

Other operations

19. Non-BlueJ packages can be opened with the *Package: Open…* command.
20. Classes can be copied into a package from outside by using the *Import Class…* command.
21. Static methods can be called from the class's popup menu.