



COMPANION MANUAL

To accompany Big Java 2e and
Java Concepts 4e by Cay Horstmann

BRUCE QUIG, Deakin University

Based on the BlueJ Tutorial by Michael Kölling



John Wiley & Sons, Inc.

SENIOR ACQUISITIONS EDITOR

Bill Zobrist

EDITORIAL ASSISTANT

Bridget Morrisey

PRODUCTION EDITOR

Marsheela Evans

MARKETING DEVELOPMENT MANAGER

Phyllis Cerys

Book printed and bound by _____ . The cover was printed by _____ .

This book is printed on acid free paper. ∞

Copyright © 2005 John Wiley & Sons, Inc. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc. 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978)646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008.

To order books or for customer service please, call 1-800-CALL WILEY (225-5945).

ISBN 0-471-73712-7

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

Contents

- 1 *About BlueJ 1***
 - About this manual 2
 - Scope and audience 2
 - Feedback 2
- 2 *Installation 3***
 - Installation on Windows 3
 - Installation on Macintosh 4
 - Installation on Linux/Unix and other Systems 4
 - Installation problems 4
- 3 *Getting started – using classes 5***
 - Starting BlueJ 5
 - Creating the project directory 5
 - Exploring System classes 5
 - Creating Library objects 7
 - Execution 8
- 4 *Writing your own classes 11***
 - Opening an existing project 11
 - Editing a class 12
 - Compilation 13
 - Help with compiler errors 14
 - Creating objects 14
 - Adding a Driver class 16
- 5 *Doing a bit more... 19***
 - Passing objects as parameters 19
 - Inspection 19
- 6 *Working with a new project 23***
 - Creating classes 23
 - Creating dependencies 23
 - Removing elements 24
- 7 *Using the code pad 25***
 - Showing the code 25
 - Simple expression evaluation 26
 - Receiving 27
 - Inspecting objects 28

| | | |
|-----------|---|-----------|
| | Executing statements | 28 |
| | Multi-line statements and sequences of statements | 28 |
| | Working with variables | 28 |
| | Command history | 29 |
| 8 | <i>Debugging</i> | 31 |
| | Setting breakpoints | 31 |
| | Stepping through the code | 33 |
| | Inspecting variables | 34 |
| | Halt and terminate | 34 |
| 9 | <i>Creating stand-alone applications</i> | 35 |
| 10 | <i>Creating Applets</i> | 37 |
| | Running an applet | 37 |
| | Creating an applet | 40 |
| | Testing the applet | 40 |
| 11 | <i>Other Operations</i> | 41 |
| | Opening non-BlueJ packages in BlueJ | 41 |
| | Adding existing classes to your project | 41 |
| | Calling static methods | 42 |
| | Generating documentation | 42 |
| | Working with libraries | 42 |
| 12 | <i>Unit testing</i> | 43 |
| | What is unit testing? | 43 |
| | Unit testing in BlueJ | 44 |
| | Enabling unit testing functionality | 44 |
| | Creating test classes | 44 |
| | Creating test methods | 46 |
| | Running tests | 48 |
| | Interpreting test results | 49 |
| | What is a fixture? | 50 |
| | Creating and using test fixtures | 50 |
| | Writing test methods by hand | 51 |
| | Writing tests first | 52 |
| | Multi-class testing | 53 |
| 13 | <i>Just the summaries</i> | 55 |
| 14 | <i>Index</i> | 59 |

Chapter 1

About BlueJ

BlueJ is an integrated Java environment specifically designed for introductory teaching. BlueJ contains and/or supports:

- fully integrated environment
- graphical class structure display
- graphical and textual editing
- built-in editor, compiler, virtual machine, debugger, etc.
- easy-to-use interface, ideal for beginners
- interactive object creation
- interactive object calls
- interactive testing
- incremental application development

The BlueJ environment was developed as part of a university research project about teaching object-orientation to beginners. The system is being developed and maintained by a joint research group at Deakin University, Melbourne, Australia, the Mærsk Institute at the University of Southern Denmark, and the University of Kent in Canterbury, UK. The project is supported by Sun Microsystems.

The aim of BlueJ is to provide an easy-to-use teaching environment for the Java language that facilitates the teaching and learning of Java. Special emphasis has been placed on visualization and interaction techniques to create a highly interactive environment that encourages experimentation and exploration.

BlueJ is implemented in Java, and regularly being tested on Solaris, Linux, Macintosh, and various Windows versions. It should run on all platforms supporting a recent Java virtual machine.

ABOUT THIS MANUAL

This manual 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, the University of Southern Denmark, Odense, and the University of Kent in Canterbury, UK.

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

SCOPE AND AUDIENCE

This manual 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. This tutorial was developed for use with Cay Horstmann's textbooks, *Big Java 2e* and *Java Concepts 4e*, both published by John Wiley and Sons.

This is not a comprehensive environment reference manual. Emphasis is on a brief and concise introduction rather than on complete coverage of features.

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 part. Chapter 13 repeats just the summary lines as a quick reference.

FEEDBACK

Comments, questions, corrections, criticisms and any other kind of feedback concerning the BlueJ system are very welcome and actively encouraged. Please mail to Michael Kölling (mik@kent.ac.uk). Likewise, comments regarding this tutorial are welcome and may be sent to Bill Zobrist (bzobrist@wiley.com).

Chapter 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.

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 2.0.2 distribution is named *bluejsetup-202.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 (eg. you have JDK 1.4.2 and JDK 1.5.0 installed), a dialogue 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 1.5.0. Changing the Java version for BlueJ will make this change for all BlueJ installations of the same version for the same user.

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 2.0.2 distribution is named *BlueJ-202.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.

INSTALLATION ON LINUX/UNIX AND OTHER SYSTEMS

The general distribution file for BlueJ is an executable jar file. It is called *bluej-xxx.jar*, where *xxx* is a version number. For example, the BlueJ version 2.0.4 distribution is named *bluej-204.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, use the distribution file *bluej-204.jar* – you need to use the file name of the file you’ve got (with the correct version number).

```
<jdk-path>/bin/java -jar bluej-204.jar
```

<jdk-path> is the directory, where JDK was installed.

A window pops up, letting you choose the BlueJ installation directory and the JDK version to be used to run BlueJ. Important: The path to BlueJ (i.e., any of the parent directories) must not contain spaces.

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

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>).

Chapter 3

Getting started – using classes

In this section we will look at performing basic tasks in BlueJ using existing classes.

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 labmanual/bank
```

CREATING THE PROJECT DIRECTORY

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

To perform most of the tasks described in this chapter, it is necessary to work from an open BlueJ project. In this case, we will create a new project. 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.

EXPLORING SYSTEM CLASSES

➤ Summary: To interact with objects from system and library classes, use *Tools – Use Library Class*.

One of the fundamental characteristics of BlueJ is that not only can you 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.

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 that cannot normally be done in Java environments.

BlueJ offers the ability to create objects from existing classes that are defined in class libraries. You can, for example, create objects of class `String` or `ArrayList`. This can be very useful for quick experimentation with these library objects. Once created, it is possible to call other methods upon these objects.

You can create a library object by selecting `Tools – Use Library Class...` from the menu. A dialogue will pop up that prompts you to enter a fully qualified class name, such as `java.lang.String`. (Note that you must type the fully qualified name, which 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 dialogue. Any of these constructors or static methods can now be invoked by selecting them from this list.

For example, if you want to create an instance of the `Rectangle` class from the `java.awt` package you enter `java.awt.Rectangle` and press `Enter`. A list of `Rectangle` constructors is shown (see **Figure 1**).

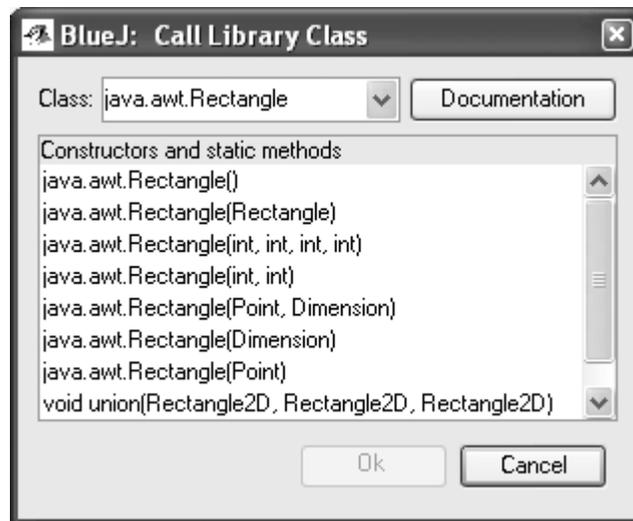


Figure 1: Creating a `java.awt.Rectangle`

From the Java API documentation we can see that the constructor that has four ints as parameters has a full signature of `Rectangle(int x, int y, int width, int height)` and creates a *Rectangle* of the specified width and height, with a top left corner at the co-ordinates of `(x, y)`.

CREATING LIBRARY OBJECTS

- Summary: To create a library object, select a constructor from the call library class dialogue.

We want to create a *Rectangle* object using the constructor shown above that takes four int parameters. Select the constructor from the list of available methods and select *OK*. You should now see a constructor dialogue where you are asked to provide the values for the four parameters (**Figure 2**).

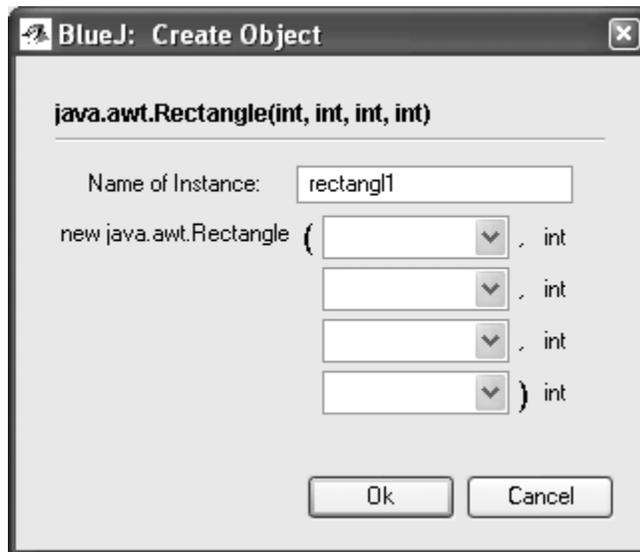


Figure 2: A Constructor dialogue for a Rectangle class constructor

This dialogue asks you for a name for the object to be created. At the same time, a default name (rectangl1) is suggested. This default name is good enough for now. We can now provide the four parameters that are required by this constructor. Provide the following values in the input areas for each parameter: 5, 5, 20, 30 and then click *OK*. A *Rectangle* object will be created which should have an initial x position of 5, y position of 5, a width of 20 and a height of 30.

Once the object has been created it is placed on the object bench (**Figure 3**). This is all there is to object creation: select a constructor from the call library dialogue, execute it, and you've got the object on the object bench. In the next sections we will call some of the available methods of our Rectangle objects.

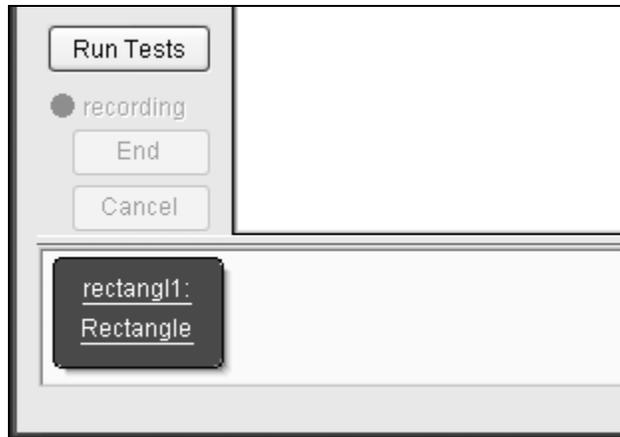


Figure 3: A Rectangle object on the Object Bench

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 Rectangle object and a menu with object operations will pop up (**Figure 4**). 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.

You see that there are many methods available that can be invoked upon a Rectangle object. We will be using two of them in particular, *getWidth()* and *translate(int, int)*. The *getWidth()* method returns a double value that represents the width of the Rectangle. This gives us an opportunity to verify that we passed in the correct information when we constructed the Rectangle. Try calling *getWidth()*. Simply select it from the object's menu and it will be executed. A dialogue appears showing you the result of the call (**Figure 5**). In this case the result returned says 20.0, which is the expected result. Optionally, you could verify the other parameters that you provided to the constructor of the Rectangle class by calling some of the other methods that are available: *getX()*, *getY()* and *getHeight()*.

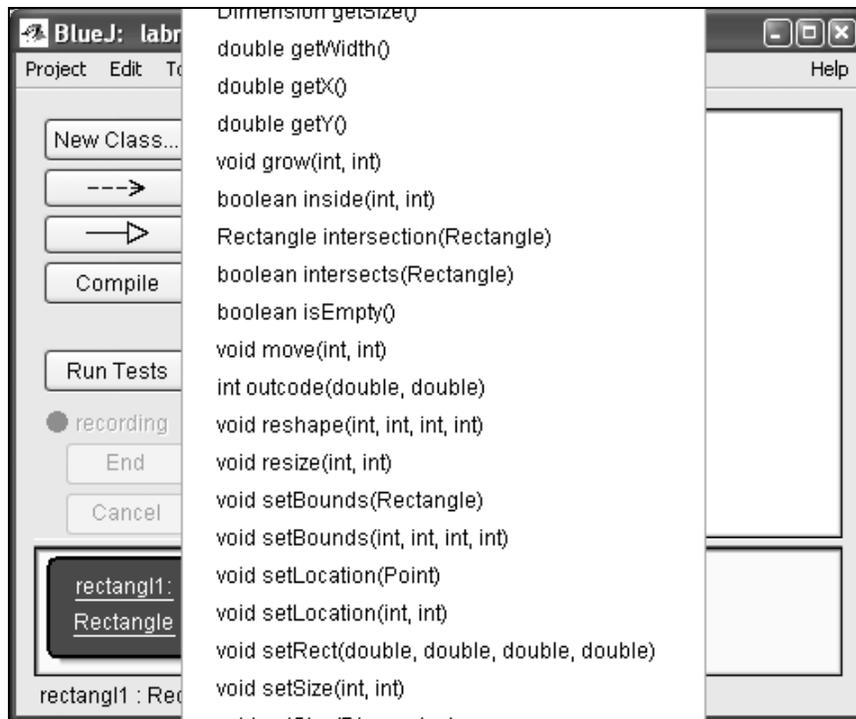


Figure 4: The object menu

Methods inherited from a superclass are available through a submenu. At the top of the object's popup menu there is one submenu for the methods inherited from the *Object* class. You can call *Object* methods (such as *toString*) by selecting them from the submenu.



Figure 5: Display of a function result

The methods we have called so far have provided information about the Rectangle object but have not changed the state of its data. Now let us try to call a method that changes some of its data. Call the method *translate* by selecting it from the menu. A dialogue appears prompting you to enter 2 parameters (**Figure 6**).



Figure 6: Function call dialogue with parameters

At the top, this dialogue shows the interface of the method to be called (including signature and comment if available). Below that is a text entry field where you can enter the parameters. The signature at the top tells us that two parameters of type `int` are expected. This method changes the Rectangle's location by moving its *x* and *y* locations by a set distance. If, for instance we wanted to move the Rectangle's *x* location by 20 units, and the *y* co-ordinate by 30 units, we would enter 20 & 30 as the parameters (in that order). We can then click *OK*.

This is all – since this method does not return a parameter there is no result dialogue. Call *getX()* and *getY()* to check that the location 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. For instance, try creating Rectangle objects using both the constructor with arguments and the default constructor which has no arguments.

Chapter 4

Writing your own classes

In the previous section we have looked at how you can interact with existing, compiled classes. In this section we will look at how you can write your own classes.

OPENING AN EXISTING 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 an existing project.

Some example projects are included with the standard BlueJ distribution in BlueJ's *examples* directory. All of the source code examples of the Big Java textbook are also available as BlueJ projects. If you are using source code examples that are provided on a CD, it will be necessary to copy these onto a local file system such as your computer's hard disk.

For this tutorial section we are using a modified version of the Bank example from the Big Java and Java Concepts text books. In this modified version we have added a Person class to help demonstrate some important features. Open the project *bank*, which is included in the labmanual directory of the Big Java source code examples. After opening the project you should see something similar to the window shown in **Figure 7**. The window might not look exactly the same on your system, but the differences should be minor.

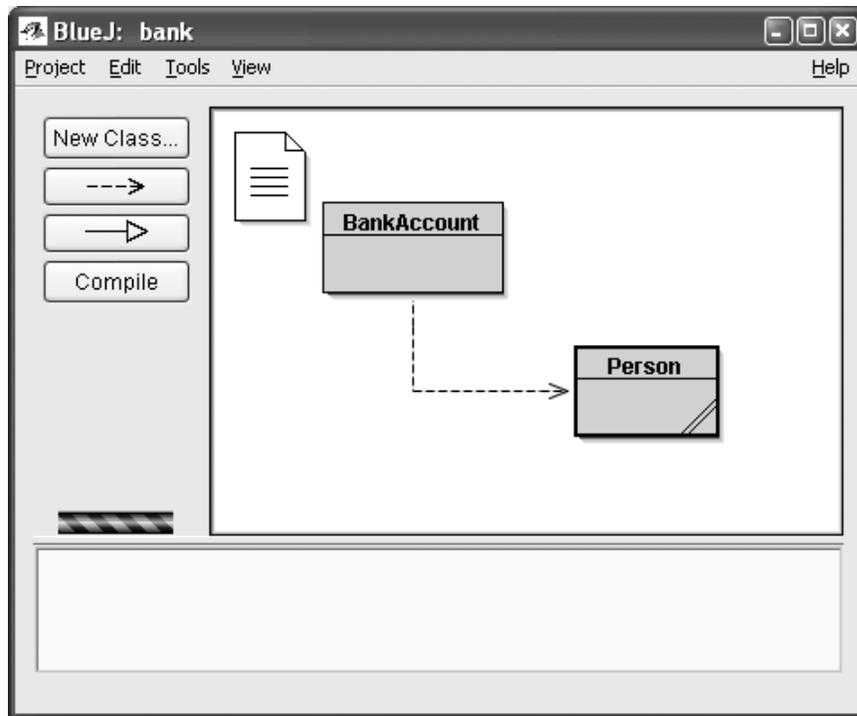


Figure 7: The BlueJ main window

The squares you see in the center part of the main window (labeled *BankAccount* and *Person*) are icons representing the classes involved in this application. These should appear as a solid color.

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 *Person* class. Find the implementation of the *getName* method. It returns, as the name suggests, the name of the *Person*. Let us change the method by adding the prefix "Name:" to the function result (so that the method returns, say, "Name: Harry" instead of just "Harry"). This assumes that the name of the person was set to Harry. This could have been done by either creating the *Person* object by using the constructor with two parameters or by using the *setName* method. We can do this by changing the line:

```

return name;
to
return "Name: " + name;

```

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

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 *Person* class has changed: it is striped now. The striped appearance marks classes that have not been compiled since the last change. You should now go back to the editor.

Side note: You may be wondering why the class icons might not have been striped when you first opened this project. This is because the classes in the bank 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 the 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 are 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 very unstable and crashes frequently, and you are worried about losing your work. Not all source code editors act the same, so be careful about saving source if using other editors.

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.

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. BlueJ tries to provide some additional help.

Open the editor again, introduce an error in the source file, and compile. In the example shown in **Figure 8** we have deliberately misspelled `yearOfBirth`. An error message should be displayed in the editor's information area. BlueJ also attempts to highlight the line of code where the error was encountered. On the right end of the information area a question mark appears that you can click to get additional information about this type of error.

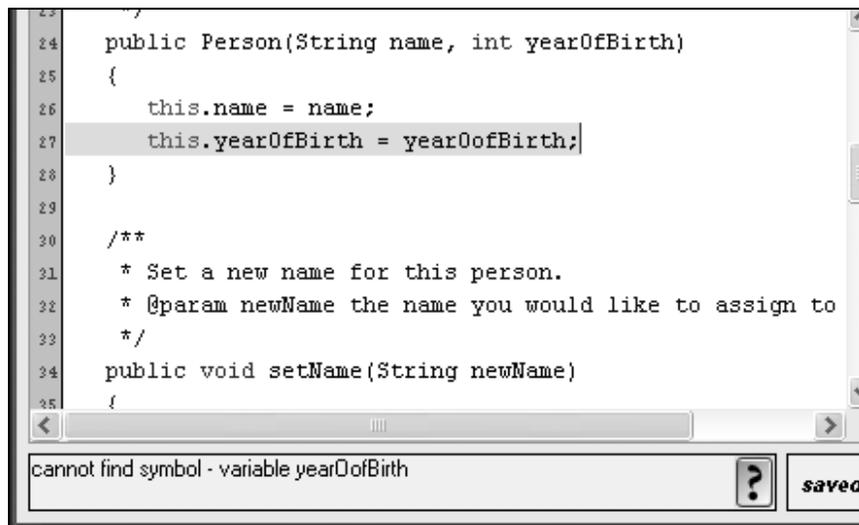


Figure 8: A compiler error and the *Help* button

At this stage, help texts are not available for all error messages. Some help texts 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. In this instance there is a spelling error in the name of the parameter being passed into the method.

CREATING OBJECTS

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

We have already seen an example of creating objects from library classes in Chapter 3, where we created an object of the library class, `java.awt.Rectangle`. We can now do the same with classes from our projects.

You can get a menu with operations applicable to a class by clicking on the class icon with the right mouse button (Macintosh: ctrl-click¹) (**Figure 9**). The operations shown are *new* operations with each of the constructors defined for this class (first) followed by some operations provided by the environment.

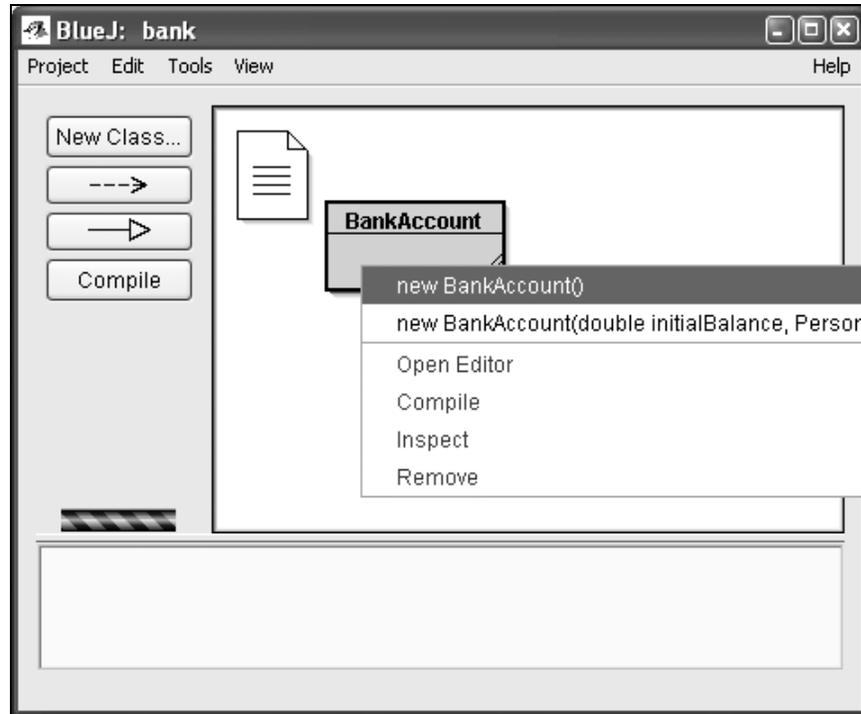


Figure 9: Class operations (popup menu)

We want to create a *BankAccount* object, so you should right-click the *BankAccount* icon (which pops up the menu shown in **Figure 9**). The menu shows two constructors to create a *BankAccount* object, one with parameters and one without. First, select the constructor without parameters. This dialogue asks you for a name for the object to be created. At the same time, a default name (*bankAcco1*) is suggested. This default name is good enough for now, so just click *OK*. A *BankAccount* object will be created (see .

You will notice that this is very similar to the way in which you created an object from a library class earlier. The main difference is the way in which you select the class and constructor. Once the object has been created it is placed on the object bench (**Figure 10**).

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

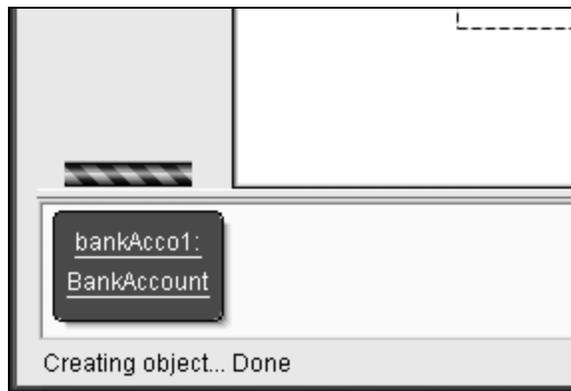


Figure 10: A BankAccount object on the object bench

ADDING A DRIVER CLASS

- **Summary:** A driver class with a main method can be created to run your application.

As we have already mentioned, one of the unique features of BlueJ is that it allows you to create individual objects and execute their methods. In most Integrated Development Environments (IDEs) you need to create an application to run any code that you have written. In Java you need to have a class that has a *main* method to create an application. This method is understood by the Java Virtual Machine as the method that starts the application. One common way of structuring an application is to use a Driver class that has a *main* method but is in no other way involved in your application.

Although a *main* method is not needed to execute code in BlueJ, there are no restrictions on you using them. They will allow you to run your code outside of BlueJ. We will now add a Driver class to the bank project.

Select the *New Class...* button or menu option in BlueJ. This will bring up a dialogue window that allows you to select the type of class you want to create and the name you would like the class to have. For our purposes select the type *Class* and type in the name *BankAccountDriver*.

A new class will appear in your main BlueJ project window. Now open the source code editor for this class. BlueJ creates some example code when you create a new class. When creating a driver class we do not need most of this code. You can delete all of the code within the class declaration. This means to delete everything in between the opening and closing braces of the class definition:

```
public class BankAccountDriver
{
}

```

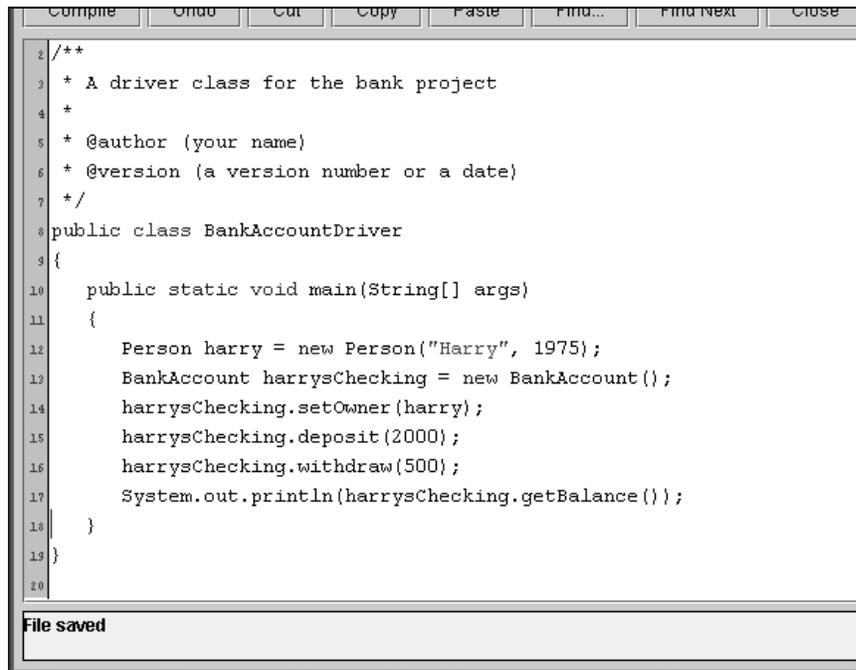
Now we can add the definition of the main method, which is

```

public static void main (String[] args)
{
}

```

We can now proceed to insert code that creates Person and BankAccount objects and call their methods.



```

1  /**
2  * A driver class for the bank project
3  *
4  * @author (your name)
5  * @version (a version number or a date)
6  */
7
8  public class BankAccountDriver
9  {
10     public static void main(String[] args)
11     {
12         Person harry = new Person("Harry", 1975);
13         BankAccount harrysChecking = new BankAccount();
14         harrysChecking.setOwner(harry);
15         harrysChecking.deposit(2000);
16         harrysChecking.withdraw(500);
17         System.out.println(harrysChecking.getBalance());
18     }
19 }
20
File saved

```

Figure 11: Driver class main method

Because this method is a static method (also commonly referred to as a class method) it appears on the popup menu of the compiled class, not on individual objects on the object bench.

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. These are commonly used to provide command-line arguments when you start up an application that provides extra data that can be used by the program. Calling the main method with the above String array would be the equivalent of the following command in a terminal window:

```
java BankAccountDriver one two three
```

In this particular instance however, we do not need any parameters so when we invoke the main method we do not need to change the default values provided by BlueJ.

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

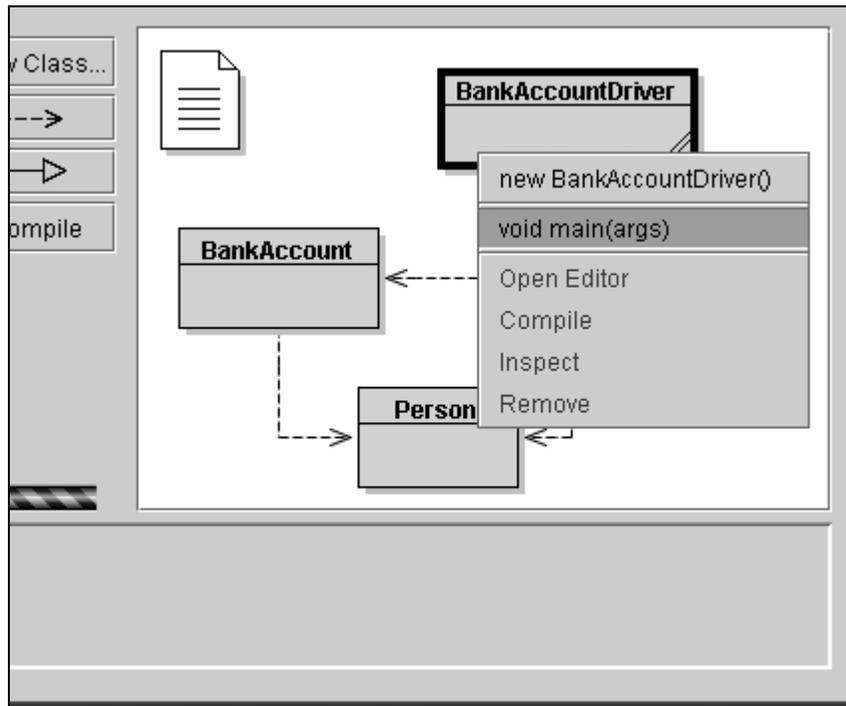


Figure 12: BankAccountDriver popup menu

Chapter 5

Doing a bit more...

In this section, we will discuss a few more things you can do in the environment. These are things that are not essential, but very commonly used.

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 provide an example. Create an object of class *BankAccount* using the constructor that has no parameters. This creates a default *BankAccount* that has no owner and a balance of zero. If you don't already have a *Person* object on the object bench, create one of those as well. For the following, you need a *BankAccount* object and a *Person* object on the object bench at the same time.

Now call the *setOwner* method of the *BankAccount* object. The signature tells you that a parameter of type *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. We can however check the result momentarily by using the *getOwner* method on the *BankAccount* object, and BlueJ's inspection features (which are described below).

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 4**). 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 *Person* object with the constructor that takes parameters). If you have just worked through the section above you should have a *Person* object already available on the Object Bench. Then select the

Inspect option from the object menu. A dialogue appears displaying the object fields, their types, and their values (**Figure 13**). The exact values shown will depend upon what method calls you have previously made.

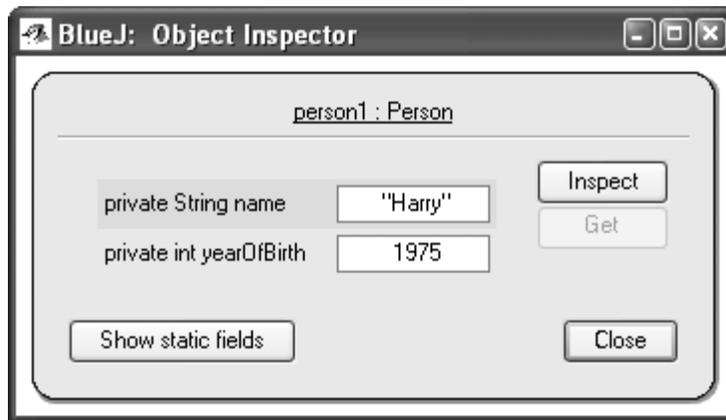


Figure 13: Inspection dialogue window

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 *Person* 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.

For the next thing that we want to try out – inspection with object fields – we can use the *BankAccount* object. Internally, the *BankAccount* code has an object of class *Person* and stores it in its *owner* field.

Now, inspect the *BankAccount* object. The resulting inspection dialogue is shown in **Figure 14**. The fields within the *BankAccount* object include an *owner* field. As you can see, its value is shown as an icon that represents an object reference – since this is a complex, user-defined object, its value cannot be shown directly in this list. To examine the owner further, select the *owner* field in the list and click the *Inspect* button in the dialogue. (You can also double-click the *owner* field.) Another inspection window is opened in turn, showing the details of the *Person* object (**Figure 15**).

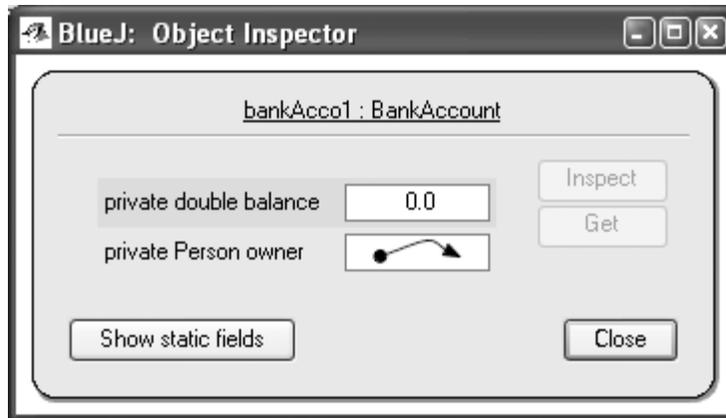


Figure 14: Inspection with object reference



Figure 15: 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.

Anytime that you invoke a method which has a return type that is of a complex object type it will appear in the result dialogue box in a similar format (shown as an object reference). It can then be inspected in the same way or placed on the Object Bench by selecting and clicking the *Get* button.

Chapter 6

Working with a new project

This chapter takes you on a quick tour of working with a new project.

CREATING CLASSES

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

In Chapter 3 we discussed how to create a new project so that we could work with library classes. As a first step to creating a new class, create a new project (select Project – New... from the main BlueJ menu).

You can now create your classes by clicking the *New Class* button on the project tool bar. This has already been briefly explained in Chapter 4 when we created a driver class for the bank project. 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 a number of different types of classes: abstract, interface, applet, “standard”, enum or unit test. 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 (e.g., 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, enum, unit test 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 of differing types and compile them. Notice how the template code for each type differs.

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 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 which case it generates an “import” statement, but we have not yet seen that 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.)

REMOVING ELEMENTS

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

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. Once again, you can also select the arrow and choose *Remove* from the arrow’s popup menu.

Chapter 7

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.

SHOWING THE CODE

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

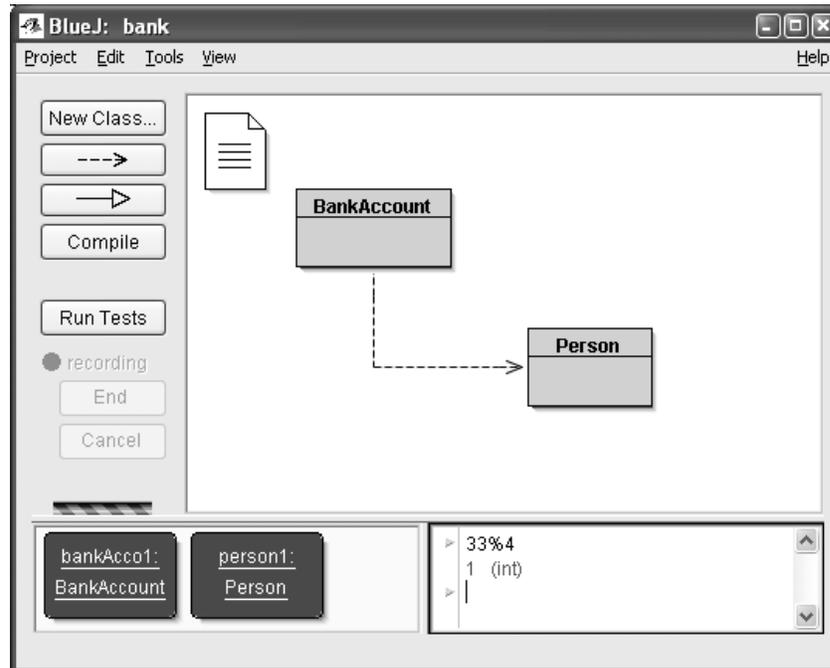


Figure 16: The main window with code pad shown

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 16**). 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.

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("Name: ")
```

Side note: When calling the `showInputDialog` method of a `JOptionPane` we are calling a method that will show a small dialog window. As this is a stand-alone dialog that is not part of a larger graphical application, this dialog has no parent component that it can use to place itself on top of. A side-effect of this in calling this method in BlueJ is that the dialog will often appear behind BlueJ's windows. If you cannot see the input dialog after calling this method, check behind BlueJ and other open application windows.

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 `BankAccount` onto the object bench (using the class popup menu as described earlier). Name it *bankAcco1* (this is the default name that BlueJ will suggest).

In the code pad, you can now type

```
bankAcco1.getBalance()
```

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

RECEIVING

- Summary: To transfer objects from the code pad to the object bench, drag the small object icon starting from the code pad, select *Show Code Pad* from the *View* menu.

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 17**).

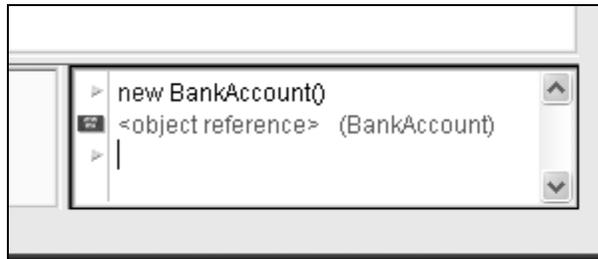


Figure 17: An object as a result of a code pad expression

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 Person()
"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 18**). 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.

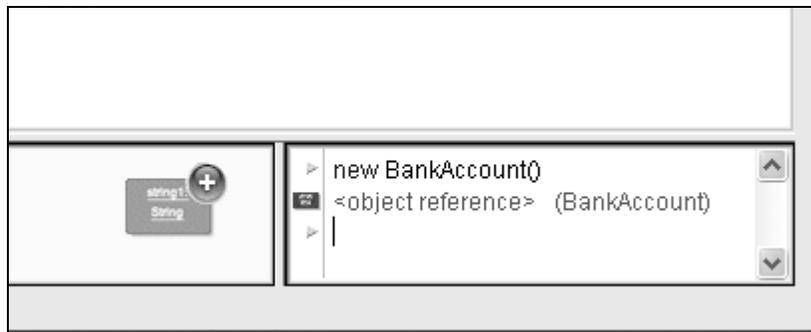


Figure 18: Dragging the object to the object bench

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.

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.

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);
}
```

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.

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.

Chapter 8

Debugging

This section introduces the most important aspects of the debugging functionality in BlueJ. In talking to computing teachers, we have 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 not enough time 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 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 *debugger*, which is included in the *labmanual* directory in the Big Java source code examples distribution.

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 19**). You can set a breakpoint by clicking into it. A small stop sign appears to mark the breakpoint. Try this now. Open the class *Word*, find the method *countSyllables*, and set a breakpoint somewhere in the *for* loop. The stop sign should appear in your editor.

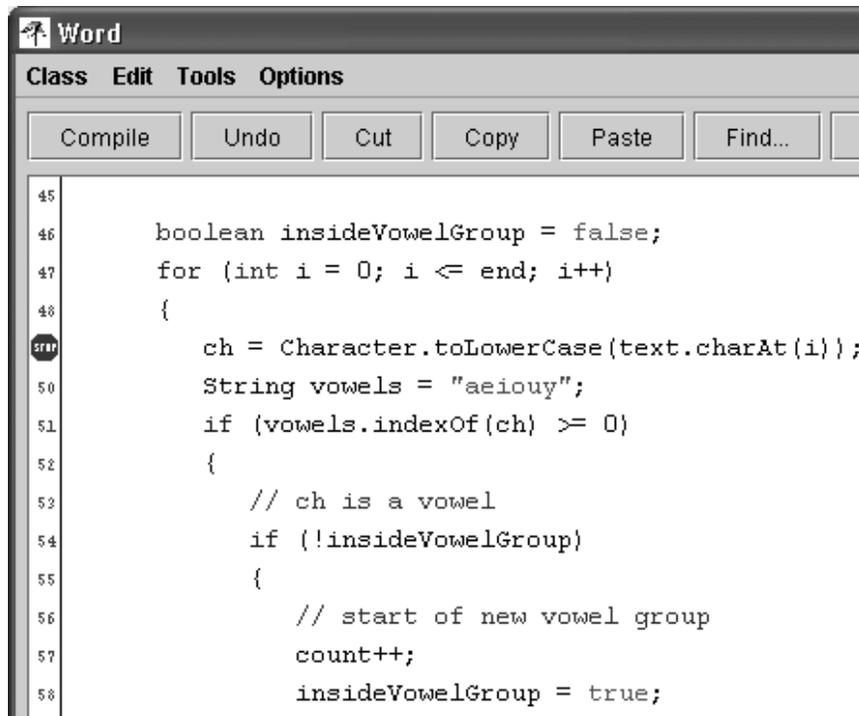


Figure 19: A breakpoint

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 *Word*. You will be asked to provide a *String* as a parameter. This will be the *String* of which the word object will try and count the syllables. A *String* like "hello" can be given. Then call the *countSyllables* method. 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 that shown in **Figure 20**.

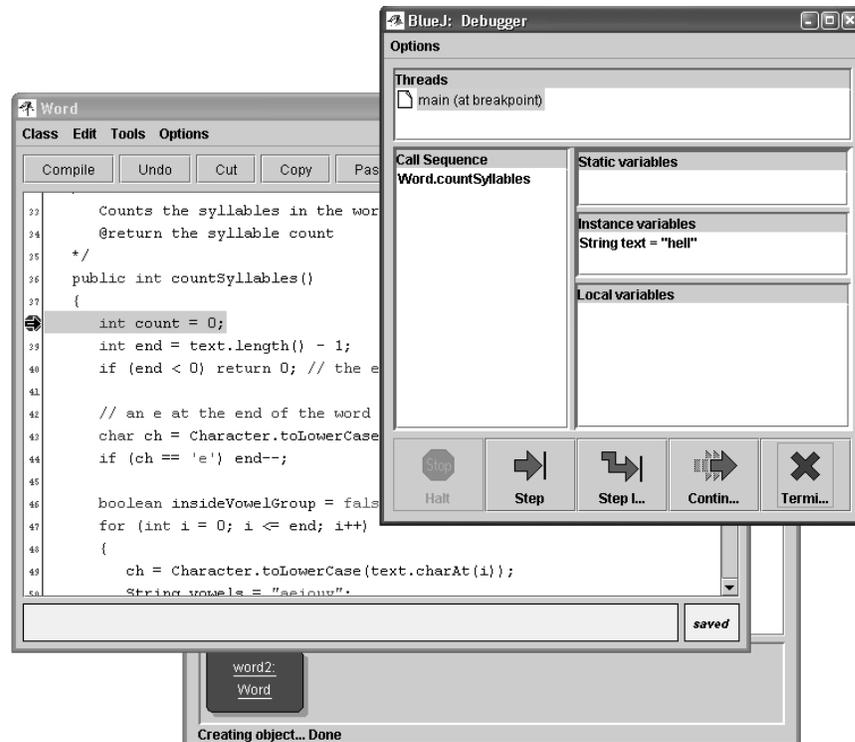


Figure 20: The debugger window

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

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 can be 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 (e.g., the local variables *count* and *end*), 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.

In addition to the *Step* command, BlueJ also provides a *Step Into* feature. 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).

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

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 – as 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.

Where methods have made calls upon other methods you can select methods in the call sequence to view variables of other currently active objects and methods. 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 (Chapter 4). There is no real difference between inspecting objects here and inspecting objects on the object bench.

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 it just takes this long. In these situations you can press the *Halt* button. The effect is similar to hitting a breakpoint set in your code. 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 (e.g., 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 or unnecessarily.

Pressing this button will terminate the whole virtual machine (VM) that is used to run code on the Object Bench and in the debugger. One side-effect is that any objects created on the Object Bench are all removed and lost, so it is advisable to use it only as an emergency mechanism.

Chapter 9

Creating stand-alone applications

- Summary: To create a stand-alone application, use *Project - Export...*

BlueJ can create executable jar files. Executable jar files can be executed on some systems by double-clicking the file (e.g., 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 *bank*. Open it (it is in the *ch02* directory). Make sure that the project is compiled. Select the *Export...* function from the *Project* menu.

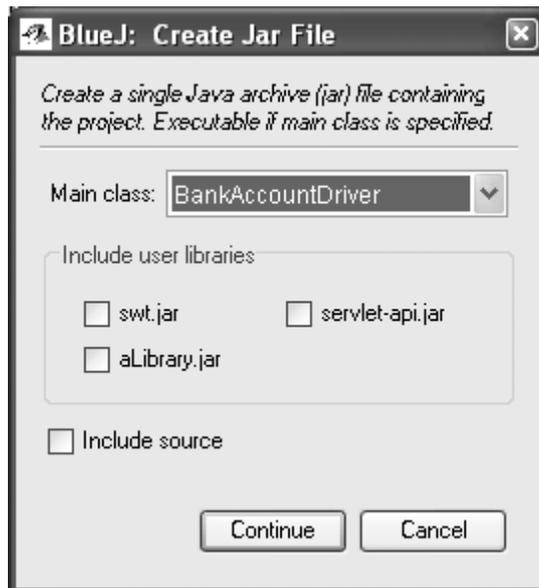


Figure 21: The "Create Jar File" dialogue

A dialogue opens that lets you specify the main class (**Figure 21**). This class must have a valid *main method* defined with the signature

```
public static void main(String[] args).
```

To complete this task you will need to have completed the earlier section on adding a Driver class. Select the *BankAccountDriver class* from the drop-down 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 if you wish, you can distribute your sources as well. 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 *bank* and click *Create*.

If you do not have libraries to be included, a file *bank.jar* will now be created. If you have libraries, a directory named *bank* will be created, and within it the jar file *bank.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 *bank.jar*). Assuming Java is installed correctly on your system, you should then be able to type

```
java -jar bank.jar
```

to execute the file.

Chapter 10

Creating Applets

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. First, we want to try executing an applet. Open the *car* project from the *labmanual* directory of the Big Java source code examples.

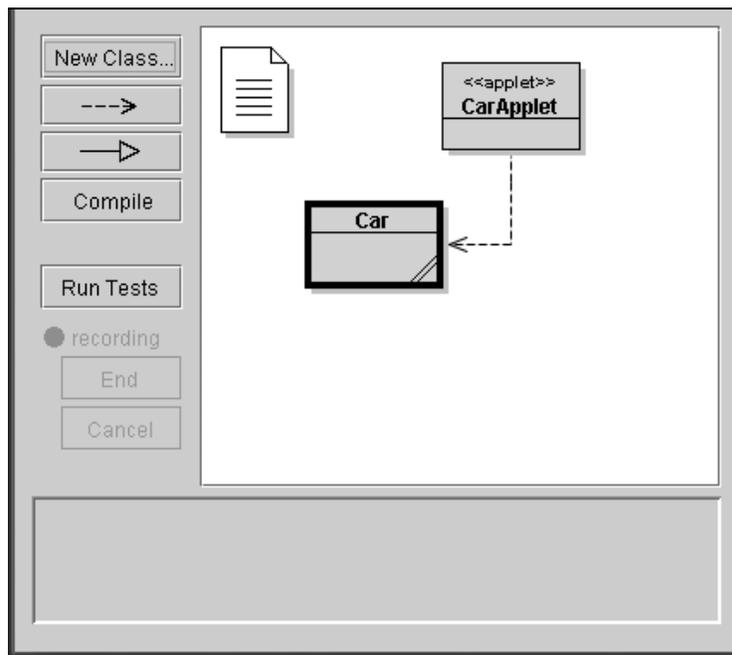


Figure 22: The car applet project

You will see that this project has two classes. The class icon of the CarApplet class is marked (with the tag <<applet>>) as an applet. After compiling, select the *Run Applet* command from this class's popup menu.

Side note: Normally to run an Applet it is necessary to not only write the code for the Applet, but to also write the html page that embeds the Applet. You would then open this web page in a java-enabled browser or use the appletviewer that comes with your Java system. In BlueJ this step is done for you when you press *Ok* in the Run Applet dialogue window. This page is re-generated every time you choose to run an Applet through BlueJ. You should not edit this html page by hand because any changes will be lost the next time you choose *Run Applet*.

A dialogue pops up that lets you make some selections (**Figure 23**).

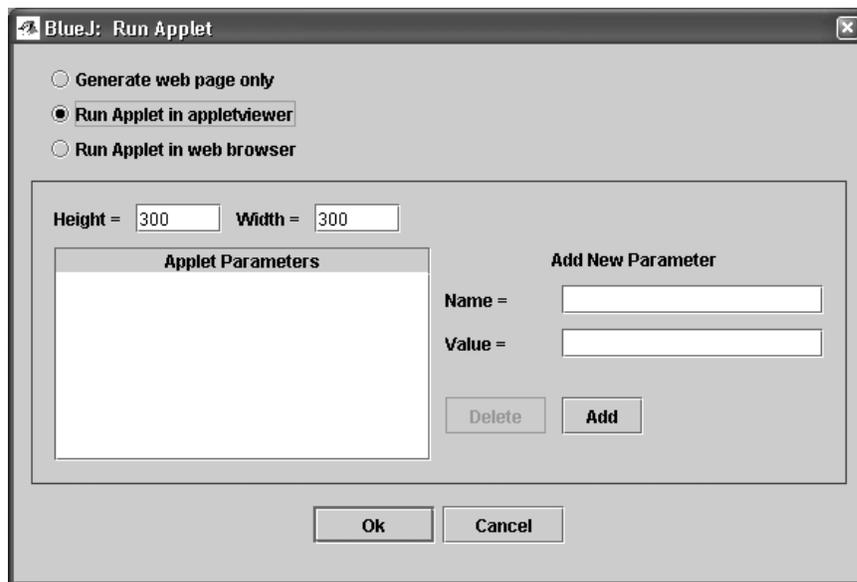


Figure 23: The "Run Applet" dialogue

You see that you have a choice of running the applet in a browser or in an appletviewer (or just to generate the web page without running it). Leave the default setting to run in appletviewer, set height and width to values somewhere around 300, and click *OK*. After a few seconds, an appletviewer similar to **Figure 24** should pop up displaying the car applet.

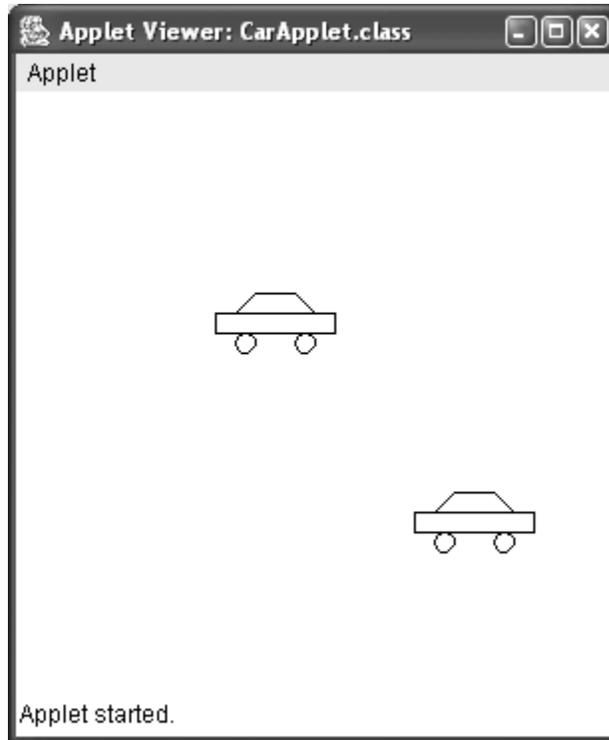


Figure 24: The Car Applet running in the appletviewer

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

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.

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.

Chapter 11

Other Operations

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.

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.

CALLING STATIC METHODS

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

We have already discussed adding a Driver class to an application in Chapter 3. Although the main method has special significance in Java applications, it is handled by BlueJ in the same way all static methods are handled.

Right-click on the class and you will see that the class menu includes not only the class's constructors, but also any static methods that are defined. You can now call these static methods directly from this menu (without first creating an object).

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.

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.

Chapter 12

Unit testing

- Summary: BlueJ provides regression testing functionality by integrating JUnit.

This tutorial introduces the unit testing functionality in the BlueJ environment. We assume that you are already familiar with BlueJ's general functionality from the earlier chapters.

We also assume that you are somewhat familiar with the idea of unit testing (or at least software testing in general). We give a few pointers in the next section.

WHAT IS UNIT TESTING?

The term *unit testing* refers to the individual testing of separate units of a software system. In object-oriented systems, these units typically are classes and methods. Thus, in our context, unit testing refers to the individual testing of methods and classes in BlueJ.

This tutorial discusses BlueJ's tools for *systematic* unit testing. If you are familiar with BlueJ's interaction features, then you know that it is easy in BlueJ to test individual methods interactively. We refer to this as *a-hoc testing*.

Adhoc testing is useful, but not good enough for systematic testing. The unit testing features in BlueJ give you the tools to record and replay tests, so that unit tests can easily be repeated later (typically after a change in the system), so that the developer can gain some confidence that recent changes have not broken existing functionality. This is known as *regression testing*.

The concepts of unit testing and regression testing are not new, but their popularity was greatly increased recently with the publication of the eXtreme programming methodology¹ (XP) and a unit testing tool for Java, *JUnit*.

JUnit is a regression testing framework written by Erich Gamma and Kent Beck. You can find the software and a lot of information about it at <http://www.junit.org>.

¹ To find out what eXtreme programming is, take a look, for example, at "*Extreme Programming Explained: Embrace Change*", Kent Beck, Addison Wesley, 1999. There are many other books available. A good online summary is at <http://www.xprogramming.com/xpmag/whatisxp.htm>

UNIT TESTING IN BLUEJ

The systematic testing tools in BlueJ are based on JUnit. Thus, some knowledge about using JUnit helps in understanding unit testing in BlueJ. We recommend reading an article about this (maybe not right now, but sometime later). Many such articles exist, and the JUnit web site is a good starting point to find them.

Unit testing in BlueJ combines BlueJ's interactive testing functionality with the regression testing of JUnit. Both testing methods are fully supported. Additionally, new functionality resulting from the combination of the two systems is available: interactive test sequences can be recorded, for example, to automatically create JUnit test methods for later regression testing. Examples are given later in this document.

The unit testing functionality in BlueJ was designed and implemented by Andrew Patterson (Monash University) as part of his PhD work.

ENABLING UNIT TESTING FUNCTIONALITY

- Summary: Testing tools can be made visible with a switch in the preferences.

The explicit testing support in BlueJ is initially disabled. To use the testing tools select Tools – Preferences... and select the checkbox labeled Show testing tools.

Enabling this functionality adds three elements to the interface: some buttons and a “recording” indicator in the main window's tool bar, a Show Test Results item in the View menu, and a Create Test Class item in the popup menu of compiled classes.

CREATING TEST CLASSES

- Summary: Create a test class by selecting *Create Test Class* from the class popup menu.

The first step in setting up testing of a class or method in BlueJ is to create a test class.

A test class is a class associated with a project class (which we will call the *reference class*). The test class contains tests for methods for the reference class.

For the examples in this tutorial, we use the *root1* project, which is distributed in the *labmanual* directory. You may wish to open it on your system to try out things as you read this.

There are two ways of creating unit test classes in BlueJ; by right-clicking a class or by choosing the “New Class” button or menu item. If you create a test class by right-clicking (MacOS: control-clicking) a compiled class and selecting the Create Test Class item from the popup menu, the test class is automatically named by adding a “Test” suffix to the name of the reference class. For example, if the reference class name is “RootApproximator”, the test class will be named “RootApproximatorTest”. The test class is then associated with the reference class and appears visually nested with the reference class. In most cases this is suitable. However, if a class of the same name exists in the package, the test class will not be created.

If you create a test class by selecting the “New Class” button or menu item and select the “Unit Test” option, you are able to specify the name of the test case to be created.

Test classes that do not follow the naming convention mentioned do not become associated with a reference class.

Test classes are shown in the diagram marked with a <<unit test>> tag attached to the reference class. They also have a different color (**Figure 25**). Dragging the reference class will keep the test class attached.

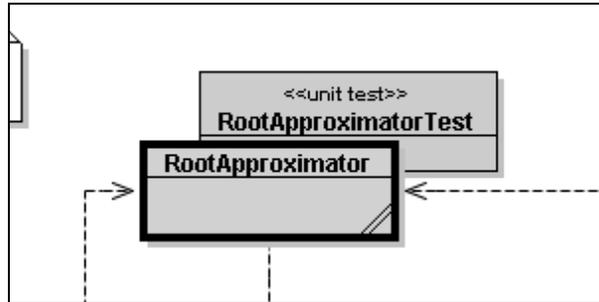


Figure 25: A reference class with an associated test class

Test classes are treated in a specialized way by the environment. They have the usual class functions (such as Open Editor, Compile, Remove), but also some test specific functions (**Figure 26**). The test class must be compiled to see this menu.

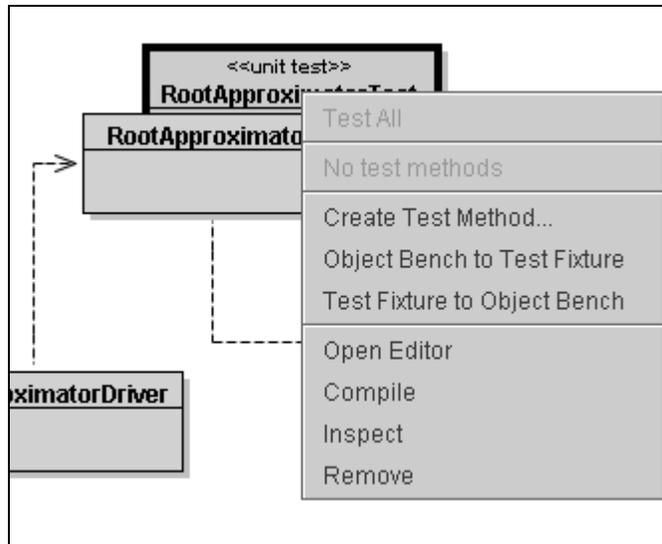


Figure 26: The popup menu of a test class

Creating the test class in itself does not create any tests, but it gives us the option of creating tests now. The test class is used to hold the test we will create.

CREATING TEST METHODS

- Summary: Create a test method by selecting *Create Test Method...* from the test class's menu.

The `RootApproximator` class contains the method `getRoot` which computes approximations to the square root of a number. Assume that we want to make a test that checks that this method works as we expect it to.

First we need to create a test class as outlined previously. You can use the popup menu of the `RootApproximator` class and select `Create Test Class`. We then select `Create Test Method...` from the `RootApproximatorTest` class. A *test method* implements a single test (i.e., the testing of one bit of functionality).

After selecting this function, you will be prompted for a name for this test. Test names always start with the prefix “test” — if your chosen name does not start with “test” this will be automatically added. Thus, typing “testGetRoot” or “getRoot” will both result in creating a test method called “testGetRoot”.

After typing the name and clicking *OK*, all interaction will be recorded as part of this test. The ‘recording’ indicator is on, and the buttons to end or cancel this test recording are enabled (**Figure 27**).

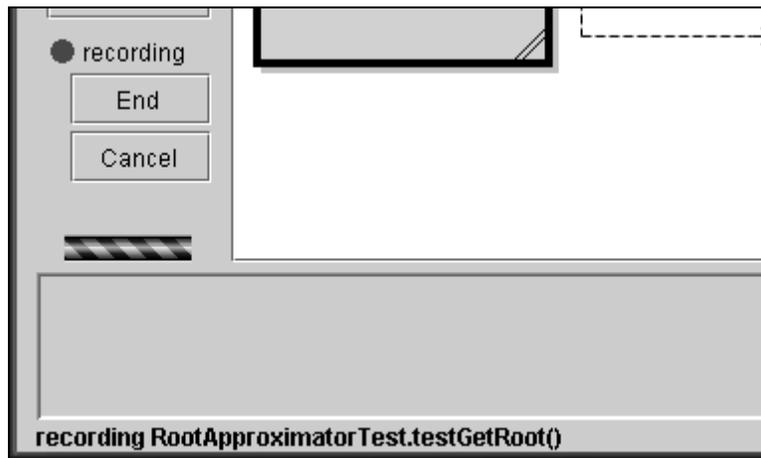


Figure 27: Test buttons during test recording

For this first example we will create a fairly simple case. We will simply test whether the method can successfully compute the root of a well-known value. To record the test in this example, do the following:

- Create a `RootApproximator` object, using the value of 100 as the parameter to the constructor.
- Call the `getRoot ()` method.

After calling the `getRoot` method, you will see the result dialogue. While we are recording tests, the result dialogue includes a part that lets us specify assertions on the result (**Figure 28**). We can use this assertion facility to specify the expected outcome of the test

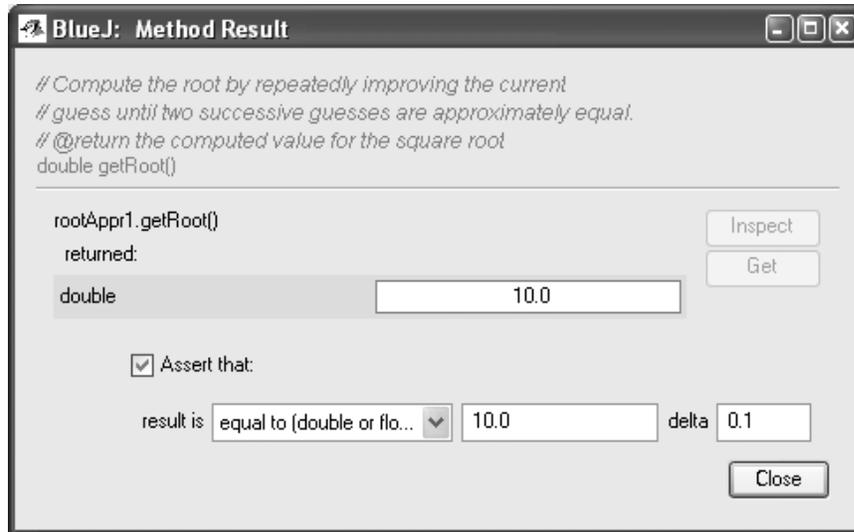


Figure 28: A result dialogue with assertion options

Several different kinds of assertions are available from the popup menu, including tests for equality, null, and not null. In our case, we expect the result of the method call to be equal to 10, so we can specify this as an assertion (**Figure 28**). Because the result of this method is a double, JUnit's underlying equality checks provide an extra parameter *delta* to allow you to express the level of accuracy you require. For this first example we have selected a delta value which is not very mathematically accurate. The level of accuracy will depend on the requirements of your application. Once we have entered the appropriate values we can close the dialogue.

This concludes our initial test case, so we can now click *End* under the test recording indicator to end the recording of the test. Ending the test will result in a test method being added to the test class. This test method is then available for running.

Side note: BlueJ will quietly add the source code to your test class and compile it. If the test class appears striped it indicates that there was a problem in compiling the new code. By re-trying to compile this class you can locate the compilation error and identify the problem code.

We can use the recording *Cancel* button to end the recording and discarding it. In a similar way to this example, we can record any number of tests. Each class in the project can have its own test class, and each test class can have any number of tests. Each test

recording can consist of an arbitrary number of actions, including arbitrary creation of instances and any number of assertions.

RUNNING TESTS

- Summary: Run all tests by clicking the *Run Tests* button. Run individual tests by selecting them from the test class's menu.

Once tests have been recorded, they can be executed. Test classes are Java classes just like the reference classes, so they too must be compiled before execution.

BlueJ automatically attempts to compile test classes after each test recording. If the assertion expression contains an error, or if the test class was edited by hand, it may be necessary to compile the test class explicitly before it can be used.

We can now right-click the test class and see the test we have just recorded in the class's popup menu. **Figure 29** shows an example with our `testGetRoot` test method.

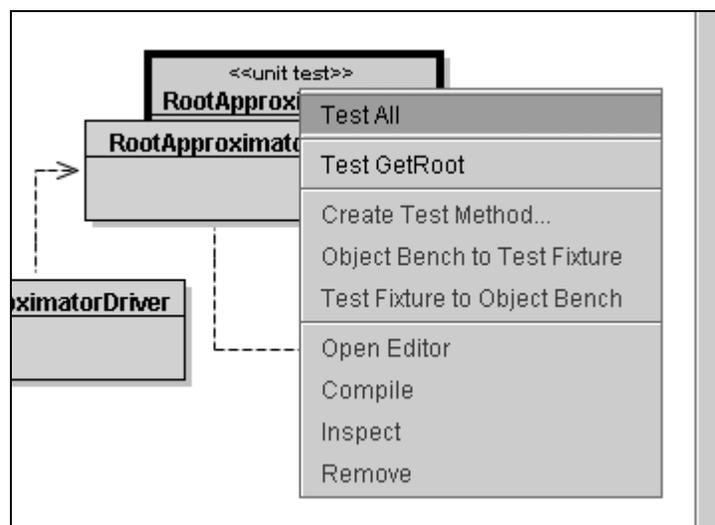


Figure 29: Test class menu with two defined test methods

Selecting a test from the menu executes that test individually. Selecting the *Test All* option from the test class's menu runs all tests defined in that class.

When a test is run individually, one of two things will happen: if the test is successful (the assertions in the test hold) a brief note indicating success is shown in the project window's status bar at the bottom of the window. If the test fails (an assertion fails or any other problem occurs) a test result window is displayed presenting details about the test run (**Figure 30**).

If all tests are run, the test result window is always displayed to show the outcome of the tests.

You can also use the *Run Tests* button above the test recording indicator in the main window. Activating this button will run all tests in all test classes in the package. This is the standard way to execute a full test suit for the package.

INTERPRETING TEST RESULTS

- Summary: The test result window shows a summary of test runs and can display failure details.

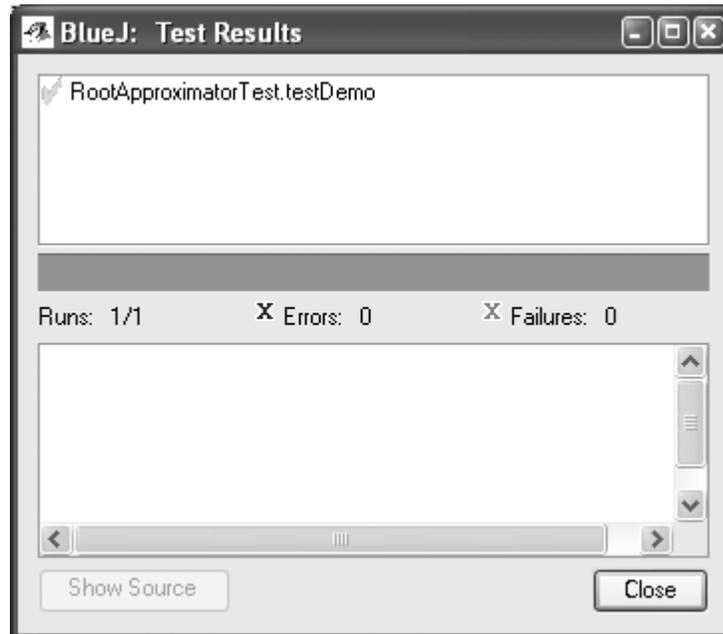


Figure 30: The test result window

When tests have been executed, the Test Result window displays a summary of the testing outcomes (**Figure 30**). The top pane of that window shows a list of all executed tests, tagged with an icon indicating their success or failure. A green tick mark indicates a successful test, a gray cross indicates a test failure, and a red cross marks an error.

The number of run tests, errors, and failures is also summarized in the middle section of the window.

A test has a failure (gray cross) if one of its assertions does not hold. For example, the test assertion may specify that a particular method result should not be null, but in this case it was.

A test has an error if its execution resulted in any other kind of error, such as an unexpected exception being thrown.

For any unsuccessful test, details about its failure can be displayed by selecting the test in the list. The lower pane in the window then displays detail information about this failure or error. The Show Source button will then be enabled. Pressing this will open the source code editor for the relevant class, highlighting the line where the error occurred.

The bar in the middle of the test window is the main summary of the test run: if it appears green, all is well — all tests have been successful. If it is red, there is a problem — at least one test has failed.

Note that on MacOS this bar does not change color.

WHAT IS A FIXTURE?

- Summary: A test fixture is a prepared set of objects used as a starting point for tests.

In our first test method we created a single object and called one of its methods. Sometimes a test needs numerous objects to conduct a test, and often there may need to be some interaction between these objects to put the system into a valid state.

We could start every individual test by creating the necessary objects and putting them into an appropriate state for doing our test. But as tests get more sophisticated, this can become tedious, and we can use a better mechanism to avoid this overhead.

We can create a state on the object bench (a set of objects, each in a certain state) which we want to use as a starting point for many or all of the tests in a specific test class. This starting set of objects is called a *fixture*.

Fixtures can be defined, and they will then automatically be set up at the start of every test of the same test class, thus reducing the overhead of each individual test.

CREATING AND USING TEST FIXTURES

- Summary: To create a fixture for a test class, create the desired objects on the object bench and then select *Object Bench To Test Fixture* from the test class's menu.

We start creating a test fixture by simply creating the objects we need, and making method calls on the objects to set them into the desired state.

For example, for testing our `RootApproximator` class in the `root1` project, we would ideally like to try more initial values that test the behavior of the `getRoot` method. Although correctly passing our test to compute the root of 100, we do not know how our method handles other values, such as (0, 0.1, 1, 4, 20, 100). We can start by simply creating a range of `RootApproximator` objects that represent this range of values.

Once the state on the object bench is what we want to start our tests, we can select the `Object Bench To Test Fixture` function from the `DatabaseTest` class.

Selecting this function will create a test fixture for the class, while removing all objects from the bench.

When a class has a fixture, this fixture will be recreated at the start of every test. For example, if we now create a new test for the `RootApproximator` class (by selecting `Create Test Method` from its test class) the situation defined in the fixture will automatically be restored. The fixture objects will appear in their defined state on the bench at the start of the test recording. We can now call the `getRoot` method of each object interactively and specify our expected results for each.

The fixture state can also be explicitly recreated on the bench by selecting `Test Fixture To Object Bench` from the test class's menu. This can be useful in cases where we want to extend the fixture later, because new test methods may need additional fixture objects.

In that case, we would use Test Fixture To Object Bench to regenerate the fixture state, and then manually make the desired additions to the fixture state. Once finished, we can again select Object Bench To Test Fixture to store the fixture. The old fixture will be replaced.

WRITING TEST METHODS BY HAND

- Summary: Test methods can be written directly in the test class's source code.

Generating test methods and fixtures by recording our interaction and the object bench state is only one option of generating unit tests. The other option is to write these test methods by hand.

Test classes are Java classes like the other classes in the project, and they can be treated the same way. In particular, we can open an editor to see the source code, we can edit the code, compile and run it.

In traditional (non-BlueJ) use of JUnit, this is the standard way of creating test methods, and BlueJ allows the same style of working. Recording tests interactively is an addition to writing tests by hand, not a replacement. There are many situations where handwriting the tests allows greater flexibility in the ways in which you can test your code.

For people unfamiliar with JUnit, a good way to start is to generate a test fixture and a few test methods interactively, and then to examine the test classes' source code. We will notice that each test class has a method named `setUp()` that is used to set up the test fixture. It also has one additional method for each test.

It is perfectly acceptable to edit existing test methods by hand to modify their behavior, or to add completely new, handwritten test methods. Remember that a test method's name must start with the prefix "test" to be recognized as a test method.

In our `RootApproximator` example we have tested equality with BlueJ's built-in equality test, which uses JUnit's `assertEquals` method at a source-code level. We may want to provide more specialized checking with the use of other classes. The `Numeric` class in the `root1` project provides sophisticated equality checks for floating point numbers. We can write a test method by hand that uses JUnit's `assertTrue` method, in conjunction with the value returned by using the `Numeric` class and its `approxEqual` method as shown in **Figure 31**.

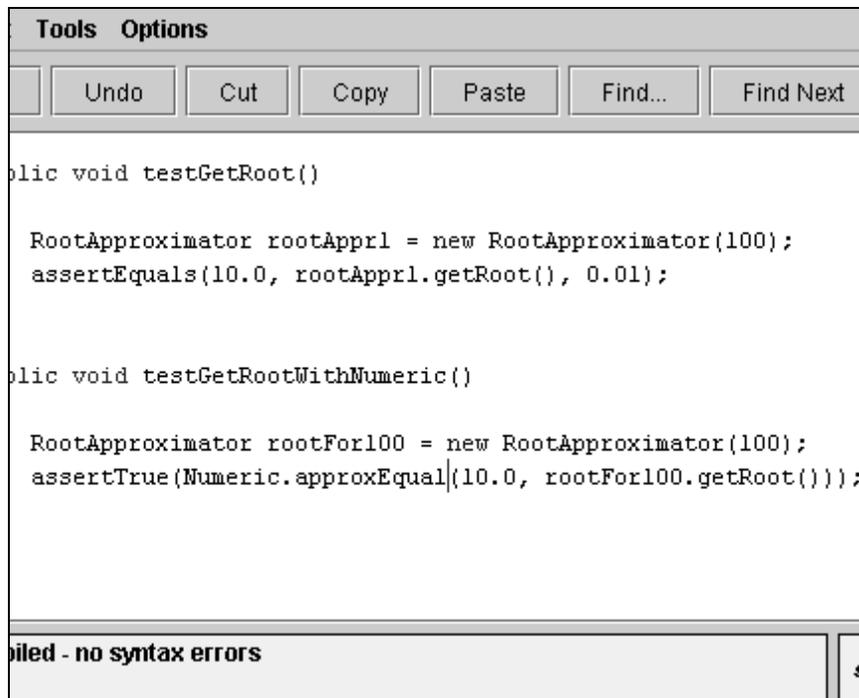


Figure 31: Handwritten test method in RootApproximatorTest

Those who want to learn more about writing JUnit tests should read some of the JUnit literature available at (<http://www.junit.org>).

WRITING TESTS FIRST

- **Summary:** To create tests before implementation, tests can be written by hand, or method stubs can be used.

The eXtreme Programming methodology suggests that tests should be written *before* the implementation of any method. Using BlueJ's unit test integration this can be done in two different ways.

First, tests can be written by hand, as explained in the previous section. Test writing then works the same way as in non-BlueJ implementations of JUnit.

Second, we can create method stubs in the reference class, returning dummy values for methods with non-void return types. After doing this, we can create tests using the interactive recording facility, and write assertions according to our expectations of the finished implementation.

MULTI-CLASS TESTING

- Summary: The *New Class...* function with class type *Unit Test* can be used to create unattached test classes.

The previous examples use test classes which are attached to a reference class. This attachment does not prevent the test classes from making use of other class types in its tests, but it suggests a logical connection of the test class to the reference class.

Sometimes test classes are written that test several classes in combination. These classes do not logically belong to a single class. To document this, they should not be directly attached to a single class.

We can create unattached test classes by using the normal *New Class...* function, and then selecting *Unit Test* as the class type in the new-class dialogue.

Unattached test classes can be used in the same way as other test classes. We can create test fixtures, make test methods, and run.

Chapter 13

Just the summaries

GETTING STARTED – USING CLASSES

- To create a project, select *New...* from the *Project* menu
- To interact with objects from system and library classes, use *Tools – Use Library Class*.
- To create a library object, select a constructor from the call library class dialogue.
- To execute a method, select it from the object popup menu.

WRITING YOUR OWN CLASSES

- To open a project, select *Open* from the *Project* menu.
- To edit the source of a class, double-click its class icon.
- To compile a class, click the *Compile* button in the editor. To compile a project, click the *Compile* button in the project window.
- To get help for a compiler error message, click the question mark next to the error message.
- To create an object, select a constructor from the class popup menu.
- A driver class with a main method can be created to run your application.

DOING A BIT MORE...

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

CREATING A NEW PROJECT

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

- To create an arrow, click the arrow button and drag the arrow in the diagram, or just write the source code in the editor.
- To remove a class, select the *Remove* function from its popup menu.
- To remove an arrow, select *Remove Arrow* from the *Edit* menu and click on the arrow.

CODE PAD

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

DEBUGGING

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

CREATING STAND-ALONE APPLICATIONS

- To create a stand-alone application, use *Project - Export...*

CREATING APPLETS

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

OTHER OPERATIONS

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

- Classes can be copied into a project from outside by using the *Add Class from File...* command.
- Static methods can be called from the class's popup menu.
- To generate documentation for a project, select *Project Documentation* from the *Tools* menu.
- The Java standard class API can be viewed by selecting *Help - Java Standard Libraries*.

UNIT TESTING

- BlueJ provides regression-testing functionality by integrating JUnit.
- Testing tools can be made visible with a switch in the preferences.
- Create a test class by selecting *Create Test Class* from the class popup menu.
- Create a test method by selecting *Create Test Method...* from the test class's menu.
- Run all tests by clicking the *Run Tests* button. Run individual tests by selecting them from the test class's menu.
- The test results window shows a summary of test runs and can display failure details.
- A test fixture is a prepared set of objects used as a starting point for tests.
- To create a fixture for a test class, create the desired objects on the object bench and then select *Object Bench To Test Fixture* from the test class's menu.
- Test methods can be written directly in the test class's source code.
- To create tests before implementation, tests can be written by hand, or method stubs can be used.
- The *New Class...* function with class type *Unit Test* can be used to create unattached test classes..

Index

A

Adding existing classes to your project, 41
Applets, 37
 creating an applet, 40
 Run Applet dialogue, 37
 running an Applet, 37
 testing the applet, 40
appletviewer, 39

B

BlueJ
 about, 1
 starting, 5
Breakpoints, 31

C

CarApplet, 38
Code pad, 25
 command history, 29
 executing statements, 28
 expression evaluation, 26
 inspecting, 28
 multi-line statements, 28
 receiving, 27
 showing, 25
 working with variables, 29
Compilation, 13
Compiler errors, 14
Creating a new project, 23
Creating classes, 23
Creating dependencies, 23
Creating objects, 7, 14
Creating the project directory, 5

D

Debugging, 31
 halt and terminate, 34
 inspecting variables, 34
 stepping, 33
Documentation, 42
Driver class, 16

E

Editing a class, 12
Execution, 8
Export dialogue, 35
Extreme programming, 43

F

Feedback, 2
Fixture, 50

I

Inspection, 19
Installation, 3
 Linux/Unix and other systems, 4
 Macintosh, 4
 problems, 4
 Windows, 3

J

JUnit, 43

L

Libraries
 working with libraries, 42

M

main method, 35

O

Objects as parameters, 20

Opening a project, 12

Opening non-BlueJ packages in BlueJ, 41

R

Removing elements, 24

Running tests, 48

S

Stand-alone applications, 35

Starting BlueJ, 5

Static methods

calling static methods, 41

Summaries, 55

System classes, 5

T

Test fixtures, 49

Test methods

writing test methods by hand, 50

Test results, 48

U

Unit testing, 43

creating test classes, 44

creating test methods, 46

enabling unit testing functionality, 44

unit testing in BlueJ, 44

what is unit testing?, 43