APPENDIX

A

Working with a BlueJ Project

A.1 Installing BlueJ

To work with BlueJ, you must install a Java Development Kit (JDK) and the BlueJ environment.

You can find the JDK and detailed installation instructions on this book's CD or at `http://www.oracle.com/technetwork/java/javase/overview/index.html`

You can find the BlueJ environment and installation instructions on this book's CD or at `http://www.bluej.org`

A.2 Opening a project

To use any of the example projects included on this book's CD, the projects must be copied to a writable disk. BlueJ projects can be opened but not executed from a CD (to execute, BlueJ needs to write to the project folder). Therefore, it does not usually work to use projects from the CD directly.

The easiest way is to copy the folder containing all of the book's projects (named *projects*) to your hard disk. After installing and starting BlueJ by double-clicking its icon, select *Open ...* from the *Project* menu. Navigate to the *projects* folder and select a project. (You can have multiple projects open at the same time.) Each project folder contains a `bluej.project` file that, when associated with BlueJ, can be double-clicked to open a project directly.

More information about the use of BlueJ is included in the BlueJ Tutorial. The tutorial is on the book's CD, and it is also accessible via the *BlueJ Tutorial* item in BlueJ's *Help* menu.

A.3 The BlueJ debugger

Information on using the BlueJ debugger may be found in Appendix F and in the BlueJ Tutorial. The tutorial is on the book's CD, and it is also accessible via the *BlueJ Tutorial* item in BlueJ's *Help* menu.

A.4 CD contents

On the CD that is included in this book, you will find the following files and directories:

Folder	Comment
bluej/	*The BlueJ system and documentation.*
bluejsetup-305.exe	*BlueJ installer for Microsoft Windows (all versions).*
BlueJ-305.zip	*BlueJ for Mac OS X.*
bluej-305.deb	*BlueJ for Debian-based systems (Debian, Ubuntu).*
bluej-305.jar	*BlueJ for other systems.*
tutorial.pdf	*The BlueJ Tutorial.*
testing-tutorial.pdf	*A tutorial introducing the testing tools.*
teamwork-tutorial.pdf	*The tutorial for use of team work tools.*
repository-setup.pdf	*Information about setting up a CVS or Subversion repository for team work support.*
ReadMe.htm	*CD documentation. Open this file in a web browser to read it. Contains CD content overview, installation instructions, and other useful pointers.*
jdk/	*Contains Java systems (JDK) for various operating systems.*
linux/	*JDK installer for Linux.*
solaris/	*JDK installer for Solaris.*
windows/	*JDK installer for Microsoft Windows (all versions).*
jdk-doc/	*Contains the Java library documentation. This is a single zip file. Copy this file to your hard disk and uncompress to use the documentation.*
projects/	*Contains all projects discussed in this book. Copy this complete folder to your hard disk before using the projects. Contains sub-folders for each chapter.*

A.5 Configuring BlueJ

Many of the settings of BlueJ can be configured to better suit your personal situation. Some configuration options are available through the *Preferences* dialog in the BlueJ system, but many more configuration options are accessible by editing the "BlueJ definitions file." The location of that file is *<bluej_home>/lib/bluej.defs*, where *<bluej_home>* is the folder where the BlueJ system is installed.[1]

Configuration details are explained in the "Tips archive" on the BlueJ web site. You can access it at

```
http://www.bluej.org/help/archive.html
```

[1] On Mac OS, the *bluej.defs* file is inside the application bundle. See the "Tips archive" for instructions how to find it.

Following are some of the most common things people like to configure. Many more configuration options can be found by reading the `bluej.defs` file.

A.6 Changing the interface language

You can change the interface language to one of several available languages. To do this, open the `bluej.defs` file and find the line that reads

```
bluej.language=english
```

Change it to one of the other available languages. For example:

```
bluej.language=spanish
```

Comments in the file list all available languages. They include at least Afrikaans, Catalan, Chinese, Czech, Danish, Dutch, English, French, German, Greek, Italian, Japanese, Korean, Portuguese, Russian, Slovak, Spanish, and Swedish.

A.7 Using local API documentation

You can use a local copy of the Java class library (API) documentation. That way, access to the documentation is faster and you can use the documentation without being online. To do this, copy the Java documentation file from the book's CD (a zip file) and unzip it at a location where you want to store the Java documentation. This will create a folder named *jdk-7-api-doc*.

Then open a web browser, and, using the *Open File ....* (or equivalent) function, open the file *index.html* inside this folder. Once the API view is correctly displayed in the browser, copy the URL (web address) from your browser's address field, open BlueJ, open the *Preferences* dialog, go to the *Miscellaneous* tab, and paste the copied URL into the field labeled *JDK documentation URL*. Using the *Java Class Libraries* item from the *Help* menu should now open your local copy.

A.8 Changing the new class templates

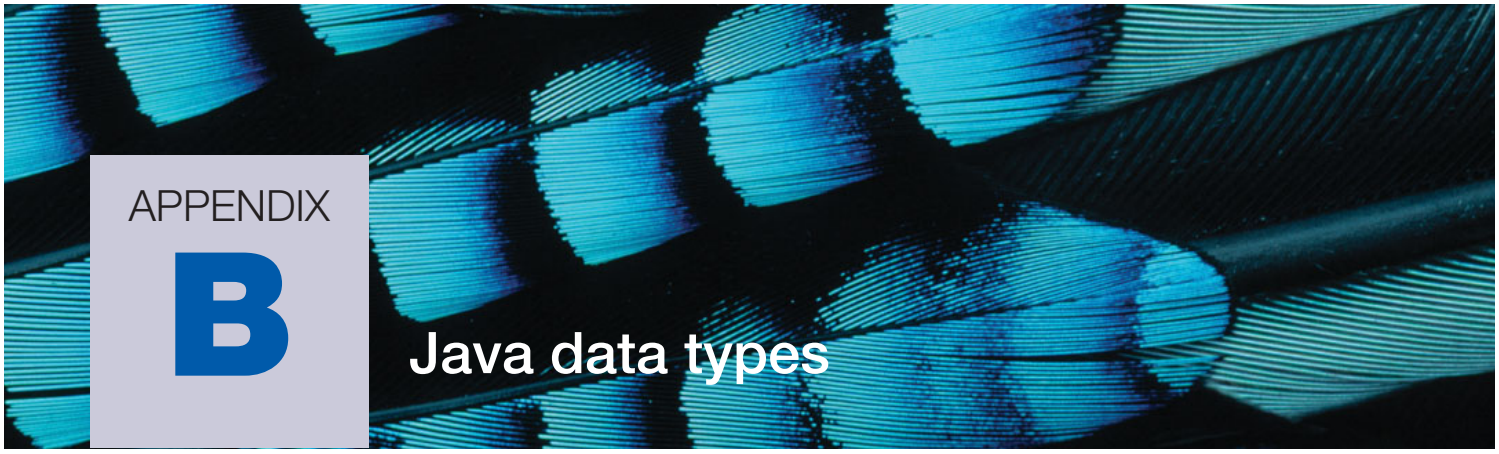
When you create a new class, the class's source is set to a default source text. This text is taken from a template and can be changed to suit your preferences. Templates are stored in the folders

<bluej_home>/lib/<language>/templates/ and

<bluej_home>/lib/<language>/templates/newclass/

where *<bluej_home>* is the BlueJ installation folder and *<language>* is your currently used language setting (for example, *english*).

Template files are pure text files and can be edited in any standard text editor.



APPENDIX

B Java data types

Java's type system is based on two distinct kinds of type: primitive types and object types.

Primitive types are stored in variables directly, and they have value semantics (values are copied when assigned to another variable). Primitive types are not associated with classes and do not have methods.

In contrast, an object type is manipulated by storing a reference to the object (not the object itself). When assigned to a variable, only the reference is copied, not the object.

B.1 Primitive types

The following table lists all the primitive types of the Java language:

Type name	Description	Example literals		
Integer numbers				
`byte`	byte-sized integer (8 bit)	`24`	`-2`	
`short`	short integer (16 bit)	`137`	`-119`	
`int`	integer (32 bit)	`5409`	`-2003`	
`long`	long integer (64 bit)	`423266353L`	`55L`	
Real numbers				
`float`	single-precision floating point	`43.889F`		
`double`	double-precision floating point	`45.63`	`2.4e5`	
Other types				
`char`	a single character (16 bit)	`'m'`	`'?'`	`'\u00F6'`
`boolean`	a boolean value (true or false)	`true`	`false`	

Notes:

- A number without a decimal point is generally interpreted as an `int` but automatically converted to `byte`, `short`, or `long` types when assigned (if the value fits). You can declare a literal as `long` by putting an `L` after the number. (`l`, lowercase `L`, works as well but should be avoided because it can easily be mistaken for a one (`1`).)
- A number with a decimal point is of type `double`. You can specify a `float` literal by putting an `F` or `f` after the number.
- A character can be written as a single Unicode character in single quotes or as a four-digit Unicode value, preceded by "\u".
- The two boolean literals are `true` and `false`.

Because variables of the primitive types do not refer to objects, there are no methods associated with the primitive types. However, when used in a context requiring an object type, autoboxing might be used to convert a primitive value to a corresponding object. See Section B.4 for more details.

The following table details minimum and maximum values available in the numerical types.

Type name	Minimum	Maximum
`byte`	`-128`	`127`
`short`	`-32768`	`32767`
`int`	`-2147483648`	`2147483647`
`long`	`-9223372036854775808`	`9223372036854775807`
	Positive minimum	**Positive maximum**
`float`	`1.4e-45`	`3.4028235e38`
`double`	`4.9e-324`	`1.7976931348623157e308`

B.2 Casting of primitive types

Sometimes it is necessary to convert a value of one primitive type to a value of another primitive type—typically, a value from a type with a particular range of values to one with a smaller range. This is called *casting*. Casting almost always involves loss of information—for example, when converting from a floating-point type to an integer type. Casting is permitted in Java between the numeric types, but it is not possible to convert a `boolean` value to any other type with a cast, or vice versa.

The cast operator consists of the name of a primitive type written in parentheses in front of a variable or an expression. For instance,

```
int val = (int) mean;
```

If `mean` is a variable of type `double` containing the value 3.9, then the statement above would store the integer value 3 (conversion by truncation) in the variable `val`.

B.3 Object types

All types not listed in Section B.1, *Primitive types*, are object types. These include class and interface types from the Java library (such as `String`) and user-defined types.

A variable of an object type holds a reference (or "pointer") to an object. Assignments and parameter passing have reference semantics (i.e., the reference is copied, not the object). After assigning a variable to another one, both variables refer to the same object. The two variables are said to be aliases for the same object. This rule applies in simple assignment between variables, but also when passing an object as an actual parameter to a method. As a consequence,

state changes to an object via a formal parameter will persist, after the method has completed, in the actual parameter.

Classes are the templates for objects, defining the fields and methods that each instance possesses.

Arrays behave like object types; they also have reference semantics. There is no class definition for arrays.

B.4 Wrapper classes

Every primitive type in Java has a corresponding wrapper class that represents the same type but is a real-object type. This makes it possible to use values from the primitive types where object types are required, through a process known as *autoboxing*. The following table lists the primitive types and their corresponding wrapper type from the `java.lang` package. Apart from `Integer` and `Character`, the wrapper class names are the same as the primitive-type names, but with an uppercase first letter.

Whenever a value of a primitive type is used in a context that requires an object type, the compiler uses autoboxing to automatically wrap the primitive-type value in an appropriate wrapper object. This means that primitive-type values can be added directly to a collection, for instance. The reverse operation—*unboxing*—is also performed automatically when a wrapper-type object is used in a context that requires a value of the corresponding primitive type.

Primitive type	Wrapper type
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`float`	`Float`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`

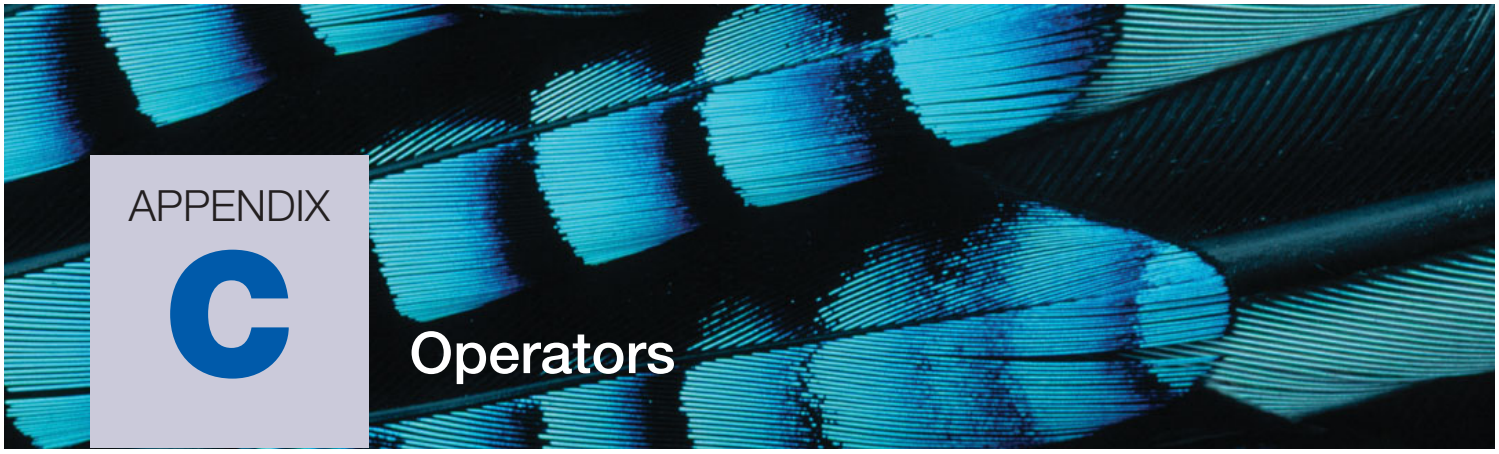
B.5 Casting of object types

Because an object may belong to an inheritance hierarchy of types, it is sometimes necessary to convert an object reference of one type to a reference of a subtype lower down the inheritance hierarchy. This process is called *casting* (or *downcasting*). The cast operator consists of the name of a class or interface type written in parentheses in front of a variable or an expression. For instance,

```
Car c = (Car) veh;
```

If the declared (i.e., static) type of variable `veh` is `Vehicle` and `Car` is a subclass of `Vehicle`, then this statement will compile. A separate check is made at runtime to ensure that the object referred to by `veh` really is a `Car` and not an instance of a different subtype.

It is important to appreciate that casting between object types is completely different from casting between primitive types (Section B.2, above). In particular, casting between object types involves *no change* of the object involved. It is purely a way of gaining access to type information that is already true of the object—that is, part of its full dynamic type.



APPENDIX

C Operators

C.1 Arithmetic expressions

Java has a considerable number of operators available for both arithmetic and logical expressions. Table C.1 shows everything that is classified as an operator, including things such as type casting and parameter passing. Most of the operators are either binary operators (taking a left and a right operand) or unary operators (taking a single operand). The main binary arithmetic operations are:

+ *addition*
− *subtraction*
* *multiplication*
/ *division*
% *modulus*, or *remainder after division*

The results of both division and modulus operations depend on whether their operands are integers or floating-point values. Between two integer values, division yields an integer result and discards any remainder, but between floating-point values, a floating-point value is the result:

`5 / 3` gives a result of 1
`5.0 / 3` gives a result of `1.6666666666666667`

(Note that only one of the operands needs to be of a floating-point type to produce a floating-point result.)

When more than one operator appears in an expression, then *rules of precedence* have to be used to work out the order of application. In Table C.1, those operators having the highest precedence appear at the top, so we can see that multiplication, division, and modulus all take precedence over addition and subtraction, for instance. This means that both of the following examples give the result `100`:

```
51 * 3 - 53
154 – 2 * 27
```

Binary operators with the same precedence level are evaluated from left to right, and unary operators with the same precedence level are evaluated right to left.

When it is necessary to alter the normal order of evaluation, parentheses can be used. So both of the following examples give the result `100`:

```
(205 – 5) / 2
2 * (47 + 3)
```

The main unary operators are –, !, ++, – –, [], and new. You will notice that ++ and – – appear in each of the top two rows in Table C.1. Those in the top row take a single operand on their left, while those in the second row take a single operand on their right.

Table C.1 Java operators, highest precedence at the top

[]	.	++	– –	(*parameters*)							
++	– –	+	–	!	~						
new	(*cast*)										
*	/	%									
+	–										
<<	>>	>>>									
<	>	>=	<=	instanceof							
==	!=										
&											
^											
|											
&&											
||											
?:											
=	+=	– =	*=	/=	%=	>>=	<<=	>>>=	&=	|=	^=

C.2 Boolean expressions

In boolean expressions, operators are used to combine operands to produce a value of either true or false. Such expressions are usually found in the test expressions of if statements and loops.

The relational operators usually combine a pair of arithmetic operands, although the tests for equality and inequality are also used with object references. Java's relational operators are:

==	equal to	!=	not equal to
<	less than	<=	less than or equal to
>	greater than	>=	greater than or equal to

The binary logical operators combine two boolean expressions to produce another boolean value. The operators are:

&&	and
||	or
^	exclusive or

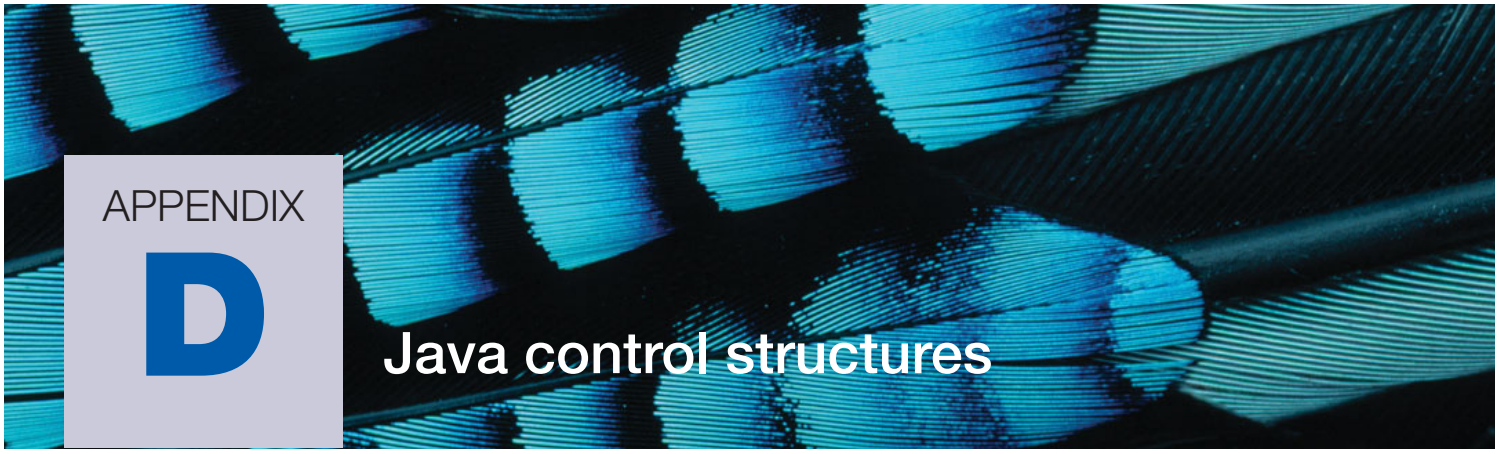
In addition,

! not

takes a single boolean expression and changes it from `true` to `false` and vice versa.

C.3 Short-circuit operators

Both `&&` and `||` are slightly unusual in the way they are applied. If the left operand of `&&` is `false`, then the value of the right operand is irrelevant and will not be evaluated. Similarly, if the left operand of `||` is `true`, then the right operand is not evaluated. Thus, they are known as short-circuit operators.



APPENDIX

D

Java control structures

D.1 Control structures

Control structures affect the order in which statements are executed. There are two main categories: *selection statements* and *loops*.

A selection statement provides a decision point at which a choice is made to follow one route through the body of a method or constructor rather than another route. An *if-else statement* involves a decision between two different sets of statements, whereas a *switch statement* allows the selection of a single option from among several.

Loops offer the option to repeat statements, either a definite or an indefinite number of times. The former is typified by the *for-each loop* and *for loop*, while the latter is typified by the *while loop* and *do loop*.

In practice, it should be borne in mind that exceptions to the above characterizations are quite common. For instance, an *if-else statement* can be used to select from among several alternative sets of statements if the *else* part contains a nested if-else statement, and a for loop can be used to loop an indefinite number of times.

D.2 Selection statements

D.2.1 if-else

The *if-else statement* has two main forms, both of which are controlled by the evaluation of a boolean expression:

```
if(expression) {
    statements
}
```

```
if(expression) {
    statements
}
else {
    statements
}
```

In the first form, the value of the boolean expression is used to decide whether or not to execute the statements. In the second form, the expression is used to choose between two alternative sets of statements, only one of which will be executed.

Examples:

```
if(field.size() == 0) {
    System.out.println("The field is empty.");
}


if(number < 0) {
    reportError();
}
else {
   processNumber(number);
}
```

It is very common to link if-else statements together by placing a second *if-else* in the *else* part of the first. This can be continued any number of times. It is a good idea to always include a final *else* part.

```
if(n < 0) {
    handleNegative();
}
else if(number == 0) {
    handleZero();
}
else {
   handlePositive();
}
```

D.2.2 switch

The *switch statement* switches on a single value to one of an arbitrary number of cases. Two possible patterns of use are:

```
switch(expression) {
    case value: statements;
                break;
    case value: statements;
                break;
    further cases possible
    default: statements;
             break;
}
```

```
switch(expression) {
     case value1:
     case value2:
     case value3:
        statements;
        break;
     case value4:
     case value5:
        statements;
        break;
     further cases possible
     default:
        statements;
        break;
}
```

Notes:

- A *switch* statement can have any number of `case` labels.
- The `break` statement after every case is needed, otherwise the execution "falls through" into the next label's statements. The second form above makes use of this. In this case, all three