Java Language Refer

Important Keywords

Data-declaration keywords:				
byte	`int`	`float`	`char`	`do`
Loop keywords:				
do	`while`	`for`	`break`	`co`
Conditional keywords:				
if	`else`	`switch`		
Exception keywords:				
throw	`try`	`catch`		
Structure keywords:				
class	`extends`	`interface`	`implements`	
Access keywords:				
public	`private`	`protected`		

Specifying Character Literals

Description or Escape Sequence	*Sequence*	*Output*
ny character	'y'	y
ackspace BS	'\b'	back space
lorizontal tab HT	'\t'	tab
ne feed LF	'\n'	linefeed
orm feed FF	'\f'	form feed
arriage return CR	'r'	carriage return
ouble quote	'\"'	"
ngle quote	'\''	'
ackslash	'\\'	\
ctal bit pattern	'0ddd'	(octal value of ddd)
ex bit pattern	'0xdd'	(hex value of dd)
nicode character	'\dddd'	(actual Unicode character of dddd

Comment Indicators

Start	Text	End Comment
/*	text	*/
/**	text	*/
//	text	(everything to the end of the line is ignored by the compiler)

Primitive Data Type Keywords

```
boolean   char   byte   short   int   long   float   double
```

Integer Data Type Ranges

Type	Length	Minimum Value	Maximum Value
byte	8 bits	–128	127
Short	16 bits	–32768	32767
Int	32 bits	–2147483648	2147483647
long	64 bits	–9223372036854775808	9223372036854775807

Unary operators

Operator	Operation
–	Unary negation
~	Bitwise complement
++	Increment
– –	Decrement
!	Not

Operator Precedence

++	– –	!	~	instanceof	*
/	%	+	–	<<	>>
>>>	<	>	<=	>=	==
!=	&	^	&&	||	?:
=	op=				

Control Statements

Statement	Example
A simple `if` statement	`if (booleanTest)` `  callfunction ( );`

A multiline `if` statement	`if (booleanTest)`
	`{`
	`  // set of statements`
	`}`

The `if...else` statement	`if (booleanTest)`
	`{`
	`  // True block statements`
	`else`
	`{`
	`  // False block statements`
	`}`

(Contd)

Statement	Example

The `while` statement	`while (booleanTest)`
	`{`
	`  // Loop statements`
	`}`

The `do...while` loop	`do`
	`{`
	`  // Loop statements`
	`}`
	`while (booleanTest);`

The `switch` statement	`switch (expression)`
	`{`
	`  case FirstCase:`
	`  // First set of statements`
	`  break ;`
	`  case SecondCase:`
	`  // Second set of statements`
	`  break ;`
	`  case ThirdCase :`
	`  // Third set of statements`
	`  break;`
	`  default :`
	`  // Default statement`
	`  break ;`
	`}`

The `for` loop	`for (initialization; condition; increment) statement;`

Defining Classes

The basic structure of defining a class is as follows:

```
Scope class ClassName [extends class]
{
    // Class implementation statements
}
```

When declaring the scope of the class, we have several options to control how other classes can access this class:

`public`	The class can be used by code outside of the file. Only one class in a file may have this scope. The file must be named with the class name followed by further four-letter .java extension.
`private`	The class can only be used within a file.
`abstract`	The class cannot be used by itself and must be subclassed.
`final`	The class cannot be used by a subclass.
`Synchronizable`	Instances of this class can be made arguments.

If a scope modifier is not used, the class is only accessible within the current file.

Defining Methods

A *method* is the code that acts on data inside a class and is always declared inside the class declaration. A method has the following syntax:

```
Scope ReturnType methodName (arguments)
{
    // Method implementation statements
}
```

The scope allows the programmer to control access to methods and can be one of the following:

`public`	The method is accessible by any system object.
`protected`	The method is only accessible by subclasses and the class in which it is declared.
`private`	The method is accessible only within current class.
`final`	The method cannot be overridden by any subclass.
`Static`	The method is shared by all instances of the class.

If a method is not given a scope, it is only accessible within the scope of the current file. We can also use thes scope operators when declaring variables.

Exception Handling

An exception has two parts: signalling an exception and setting up an exception handler. To signal an exceptio use the **try** keyword. To set up an exception handler, we use the **catch** keyword. We use the **finally** keyword specify a block of statements that will execute no matter what. To tell the system that an error has occurred, u the **throw** keyword.

```
try
{
        / / Try this block of code and throw exception
{
catch (Exception e)
{
        / / Handle error
}
finally
{
        / / Executed no matter what happens
}
```

General Applet Construction

A minimal Java Applet has the following construction:

```
/*
*  JavaApplet.java - Sample Applet *
*/
import java.applet.*;
import java.awt.Graphics;
public class JavaApplet extends java.applet.Applet
{
   public void init ( )
   {
      / / Called first time applet is executed
   }
   public void start ( )
   {
      / / Called after init( ) and whenever Web page is revisited
   }
   public void stop ( )
   {
      / / Called when Web page disappears
   }
   public void destroy ( )
   {
      //Called when applet is being removed from memory
   }
   public void paint (Graphics g)
   {
      g.drawstring ("Goodbye !", 100, 100) ;
```

APPENDIX

B

Java Keywords

This appendix lists the keywords in Java. They are grouped according to their meaning/function.

Group	*Keyword*	*Meaning/Function*
Class Organization	`package`	specifies the class in a particular source file should belong to the named package.
	`import`	requests the named class or classes be imported into the current application.
Class Definition	`interface`	defines global data and method signatures that can be shared among classes.
	`class`	defines a collection of related data behaviour.
	`extends`	indicates which class to subclass.
	`implements`	indicates the interface for which a new class will supply methods.
Keywords for Classes and Variables	`abstract`	specifies the class cannot be instantiated directly.
	`public`	means the class, method, or variable can be accessed from anywhere.
	`private`	means only the class defining the method or variable can access it.
	`protected`	means only the defining class and its subclasses can access the method or variable.
	`static`	specifies a class method or variable.
	`synchronized`	indicates only one object or class can access this variable or method at a time.
	`volatile`	tells the compiler this variable may change asynchronously due to threads.
	`final`	means this variable or method cannot be changed by subclasses.
	`const`	means this variable cannot be changed.
	`native`	links a method to native code.
Simple Data Types	`long`	is a 64-bit integer value.
	`int`	is a 32-bit integer value.
	`short`	is a 16-bit integer value.

(Cont

(Contd)

Group	Keyword	Meaning/Function
	byte	is a 8-bit integer value.
	double	is a 64-bit floating-point value.
	float	is a 32-bit floating-point value.
	char	is a 16-bit Unicode character.
	boolean	is a true or false value.
	void	indicates a method does not return a value.
Values and Variables	false	is a Boolean value.
	true	is a Boolean value.
	this	refers to the current instance in an instance method.
	super	refers to the immediate superclass in an instance method.
	null	represents a nonexistent instance.
Exception Handling	throw	throws an exception
	throws	throws an exception.
	try	marks a stack so that if an exception is thrown, it will unwind to this point.
	catch	catches an exception.
	finally	says execute this block of code regardless of exception error handling flow.
Instance Creating and Testing	new	creates new instances.
	instanceof	tests whether an instance derives from a particular class or interface.
Control Flow	switch	tests a variable.
	case	executes a particular block of code according to the value tested in the switch.
	default	means the default block of code executes if no matching case statement was found.
	break	breaks out of a particular block of code.
	continue	continues with the next iteration of a loop.
	goto	directs control to a specified place.
	return	returns from a method, optionally passing back a value.
	do	performs some statement or set of statements.
	if	tests for a condition and performs some action if true.
	else	performs some action if the above test was false.
	for	signifies iteration.
	while	performs some action while a condition is true.

- Keywords not available from C
  `auto, enum, extern, register, signed, sizeof, struct, typedef, union, unsigned.`
- Keywords not available from C++
  `delete, friend, inline, mutable, template, using, virtual.`

APPENDIX

C

Differences Between Java and C/C++

C.1 DATA TYPES

- All Java primitive data types (char, int, short, long, byte, float, double and boolean) have specified sizes and behaviour that are machine-independent.
- Conditional expressions can only be Boolean, not integral.
- Casting between data types is much more controlled in Java. Automatic conversion occurs only when there is no loss of information. All other casts must be explicit.
- Java supports special methods to convert values between class objects and primitive types.
- Composite data types are accomplished in Java using only classes. Structures and unions are not supported.
- Java does not support **typedef** keyword.
- All non-primitive types can only be created using **new** operator.
- Java does not define the type modifiers **auto, extern, registor, signed,** and **unsigned**.

C.2 POINTERS

- Java does not support pointers. Similar functionality is accomplished by using implicit references to objects. Pointer arithmetic is not possible in Java.

C.3 OPERATORS

- Java adds a new right shift operator > > > which inserts zeros at the top end.
- The + operator can be used to concatenate strings.
- Operators overloading is not possible in Java.
- The , operator of C has been deleted.

- Java adds another operator **instanceof** to identify objects.
- The modulo division may be applied to float values in Java which is not permitted in C/C++.

C.4 FUNCTIONS AND METHODS

- All functions are defined in the body of the class. There are no independent functions.
- The functions defined inside a class are known as methods.
- Although function overloading in Java works virtually identical to C++ function overloading, there are no default arguments to functions.
- No inline functions in Java.
- Java requires that methods with no arguments must be declared with empty parenthesis, (not with **void** keyword).

C.5 PREPROCESSOR

- Java does not have a preprocessor, and as such, does not support **#define** or macros.
- Constants can be created using the **final** modifier when declaring class and instance variables.
- Java programs do not use header files.

C.6 CLASSES

- Class definitions take the similar form in Java as in C++, but there is no closing semicolon.
- There is no scope resolution operator :: in Java.
- No forward references of classes are necessary in Java.
- No destructors in Java.
- Java has no templates.
- No nested classes in Java.
- Inheritance in Java has the same effect as in C++, but the syntax is different.
- Java does not provide direct support for multiple inheritance. We can accomplish multiple inheritance by using interfaces.
- Access specifiers (public, private, protected and private protected) are placed on each definition for each member of a class.
- A class in Java can have an access specifier to determine whether it is visible outside the file.
- There is no **virtual** keyword in Java. All non-static methods always use dynamic binding.
- Initialization of primitive class data member is guaranteed in Java. We can initialize them directly when we define them in the class, or we can do it in the constructor.
- We need not externally define storage for **static** members like we do in C++.

C.7 STRINGS

- Strings in C and C++ are arrays of characters, terminated by a null character. But strings in Java are objects. They are not terminated by a null. Therefore, strings are treated differently in C++ and Java.
- Strings can be concatenated using + operator.

C.8 ARRAYS

- Arrays are quite different in Java. Array boundaries are strictly enforced. Attempting to read past the end of an array produces an error.
- One array can be assigned to another in Java.
- Java does not support multidimensional arrays as in C and C++. However, it is possible to create arrays of arrays to represent multidimensional arrays.

C.9 CONTROL FLOW

- The test expressions for control flow constructs return a Boolean value (true or false) in Java. In C and C++, they return an integer value.
- The control variable declared in **for** loop is not available after the loop is exited in Java.

C.10 COMMAND-LINE ARGUMENTS

- The command line arguments passed from the system into a Java program differ in a couple of ways compared to that of C++ program.
- In C and C++, two arguments are passed. One specifies the number of arguments and the other is a pointer to an array of characters containing the actual arguments. In Java, a single argument containing an array of strings is passed.
- The first element in the arguments vector in C and C++ is the name of the program itself. In Java, we do not pass the name of the program as an argument. We already know the name of the program because it is the same name as the class.

C.11 OTHER DIFFERENCES

- Java supports multithreading.
- Java supports automatic garbage collection and makes a lot of programming problems simply vanish.
- The destructor function is replaced with a finalize function.
- Exception handling in Java is different because there are no destructors. A **finally** clause is always executed to perform necessary cleanup.
- Java has built-in support for comment documentation, so the source code file can also contain its own documentation.

APPENDIX

D

Bit-level Programming

D.1 INTRODUCTION

One of the unique features of Java language as compared to other high-level languages is that it allows direct manipulation of individual bits within a word. Bit-level manipulations are used in setting a particular bit or group of bits to 1 or 0. They are also used to perform certain numerical computations faster. As pointed out in Chapter 5, Java supports the following operators:

1. Bitwise logical operators
2. Bitwise shift operators
3. One's complement operator

All these operators work only on integer type operands.

D.2 BITWISE LOGICAL OPERATORS

There are three logical Bitwise operators. They are:

- Bitwise AND (&)
- Bitwise OR (|)
- Bitwise *exclusive* OR (^)

These are binary operators and require two integer-type operands. These operators work on their operands bit by bit starting from the least significant (i.e. the rightmost) bit, setting each bit in the result as shown in Table D.1.

Table D.1 *Java Milestones*

op1	*op2*	*op1 & op2*	*op1 | op2*	*op1 ^ op2*
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Bitwise AND

The bitwise AND operator is represented by a single ampersand (&) and is surrounded on both sides by integer expressions. The result of ANDing operation is 1 if both the bits have a value of 1; otherwise it is 0. Let us consider two variables **x** and **y** whose values are 13 and 25. The binary representation of these two variables are

```
x → 0000 0000 0000 1101
y → 0000 0000 0001 1001
```

If we execute statement

```
z = x & y;
```

then the result would be:

```
z → 0000 0000 0000 1001
```

Although the resulting bit pattern represents the decimal number 9, there is no apparent connection between the decimal values of these three variables. Program D.1 shows how to use the bitwise operators.

Program D.1 *Demonstration of bitwise operators*

```
Class Bitwise
{
   public static void main (String args[])
   {
      int a=13, b=25;
      System.out.println ("a = " + a);
      System.out.println ("b = " + b);
      System.out.println ("a & b = " + (a & b) );
      System.out.println ("a | b = " + (a | b) );
      System.out.println ("a ^ b = " + (a ^ b) );
   }
}
```

The output would be:

```
a = 13
b = 25
a & b = 9
a | b = 29
a ^ b = 20
```

Bitwise ANDing is often used to test whether a particular bit is 1 or 0. For example, the following program tests whether the fourth bit of the variable **flag** is 1 or 0.

```
Class Bit1
{
     Static final TEST = 8;                    /* represents 00....01000 */
     public static void main (String args[])
     {
          int flag;
          ..........
          ..........
          if ( (flag & TEST) ! = 0)            /* test 4th bit */
          {
```

```
                System.out.println ("Fourth bit is set \n") ;
            }
            ..........
            ..........
        }
    }
```

Note that the bitwise logical operators have lower precedence than the relational operators and therefore additional parentheses are necessary as shown above.

The following program tests whether a given number is odd or even.

```
Class Bit2
{
    public static void main (String args [])
    {
        int test = 1;
        int number;
        // Input a number here
            ..........
            ..........
        while (number ! = -1)
        {
            if (number & test)
                System.out.println ("Number is odd\n\n");
            else
                System.out.println ("Number is even\n\n");
            // Input a number here
            ..........
            ..........
        }
    }
}
```

Output:

```
Input a number
20
Number is even
Input a number
9
Number is odd
Input a number
-1
```

Bitwise OR

The bitwise OR is represented by the symbol | (vertical bar) and is surrounded by two integer operands. The result of OR operation is 1 if *at least* one of the bits has a value of 1; otherwise it is zero. Consider the variables x and y discussed above.

```
x      →   0000 0000 0000 1101
y      →   0000 0000 0001 1001
x|y    →   0000 0000 0001 1101
```

The bitwise inclusion OR operation is often used to set a particular bit to 1 in a flag. Example:

```
class Bit3
{
    final static SET = 8;
    public static void main (String args[])
    {
        int flag;
        ..........
        ..........
        flag = flag | SET;
        if ((flag & SET) ! = 0)
        {
            System.out.println ("flag is set \n");
        }
        ..........
        ..........
    }
}
```

The statement

```
flag = flag | SET;
```

causes the fourth bit of flag to set 1 if it is 0 and does not change it if it is already 1.

Bitwise Exclusive OR

The bitwise *exclusive* OR is represented by the symbol ^. The result of exclusive OR is 1 if *only one* of the bits is 1; otherwise it is 0. Consider again the same variables **x** and **y** discussed above.

```
x     →   0000 0000 0000 1101
y     →   0000 0000 0001 1001
x^y   →   0000 0000 0001 1101
```

D.3 BITWISE SHIFT OPERATORS

The shift operators are used to move bit patterns either to the left or to the right. The shift operators are represented by the symbols < and > and are used in the following form:

```
Left shift:  op << n
Right shift: op >> n
```

op is the integer expression that is to be shifted and *n* is the number of bit positions to be shifted.

The left-shift operation causes all the bits in the operand *op* to be shifted to the left by *n* positions. The leftmost *n* bits in the original bit pattern will be lost and rightmost *n* bits positions that are vacated will be filled with 0s.

Similarly, the right-shift operation causes all the bits in the operand *op* to be shifted to the right by *n* positions. The rightmost *n* bits will be lost. The leftmost *n* bit positions that are vacated will be filled with zero, if the *op* is a *positive integer*. If the variable to be shifted is *negative*, then the operation preserves the high-order bit of 1 and shifts only the lower 31 bits to the right.

Both the operands *op* and *n* can be constants or variables. There are two restrictions on the value of *n*. It may not be negative and it may not exceed the number of bits used to represent the left operand *op*.

Let us suppose **x** is a positive integer whose bit pattern is

```
0100    1001    1100    1011
```

then,

```
                                        Vacated
                                        positions
                                      ↙
x << 3  =  0100  1110  0101  1000
x >> 3  =  0000  1001  0011  1001
             ↑
          Vacated
          positions
```

Shift operators are often used for multiplication and division by powers of two. Consider the following statement:

```
x = y << 1;
```

This statement shifts one bit to the left in y and then the result is assigned to x. The decimal value of x will be the value of y multiplied by 2. Similarly, the statement

```
x = y >> 1;
```

shifts y one bit to the right and assigns the result to x. In this case, the value of x will be the value of y divided by 2.

Java supports another shift operator >>> known as zero-fill-right-shift operator. When dealing with positive numbers, there is no difference between this operator and the right-shift operator. They both shift zeros into the upper bits of a number. The difference arises when dealing with negative numbers. Note that negative numbers have the high-order bit set to 1. The right-shift operator preserves the high-order bit as 1. The zero-fill-right-shift operator shifts zeros into all the upper bits, including the high-order bit, thus making a negative number into positive. Program D.2 demonstrates the use of shift operators.

Program D.2 *Demonstration of Shift operators*

```
Class Shift
{
    public static void main (String args[])
    {
        int a=8, b=-8;
        System.out.println ("a = " + a + " b = " + b);
        System.out.println ("a >> 2 = " + (a >> 2));
        System.out.println ("a << 1 = " + (a << 1));
        System.out.println ("a >>> 1 = " + (a >>> 1));
        System.out.println ("b >> 1 = " + (b >> 1));
        System.out.println ("b >>> 1 = " + (b >>> 1));
    }
}
```

The output would be:

```
a = 8     b = - 8
a >> 2      = 2
a << 1      = 16
a >>> 1     = 4
b >> 1      = - 4
b >>> 1     = 2147483644
```

D.4 BITWISE COMPLEMENT OPERATORS

The complement operator ~ (also called the one's complement operator) is an unary operator and inverts all the bits represented by its operand. That is, 0s become 1s and 1s become zero. Example:

```
x    =   1001 0110 1100 1011
~x   =   0110 1001 0011 0100
```

This operator is often combined with the bitwise AND operator to turn off a particular bit. For example, the statements

```
x    = 8;          /* 0000 0000 00000 1000 */
flag = flag & ~x;
```

would turn off the fourth bit in the variable **flag.**

APPENDIX **E**

Java API Packages

Java API is implemented as packages, which contain groups of related classes. Along with classes, they also include interfaces, exception definitions and error definitions. Java API is composed of a large number of packages. The most commonly used packages are:

Stand-alone Application Programming

1. java.lang
2. java.util
3. java.io

Applet and Network Programming

4. java.awt
5. java.applet
6. java.net

This appendix lists the frequently used interfaces and classes contained in the above packages.

Table E.1 *Java.lang Package*

Interfaces	
`Cloneable`	Interface indicating that an object may be copied or cloned
`Runnable`	Methods for classes that want to run as threads
Classes	
`Boolean`	Object wrapper for boolean values
`Byte`	Object wrapper for byte values
`Character`	Object wrapper for char values
`Class`	Run-time representations of classes
`ClassLoader`	Abstract behaviour for handling loading of classes

(Contd)

Compiler	System class that gives access to the Java compiler
Double	Object wrapper for double values
Float	Object wrapper for float values
Integer	Object wrapper for int values
Long	Object wrapper for long values
Math	Utility class for math operations
Number	Abstract superclass of all number classes (Integer, Float, and so on)
Object	Generic object class, at top of inheritance hierarchy
Process	Abstract behaviour for processes such as those spawned using methods in the System class
Runtime	Access to the Java runtime
SecurityManager	Abstract behaviour for implementing security policies
String	Character strings
StringBuffer	Mutable strings
System	Access to Java's system-level behaviour, provided in a platform independent way
Thread	Methods for managing threads and classes that run in threads
ThreadDeath	Class of object thrown when a thread is asynchronously terminated
ThreadGroup	A group of threads
Throwable	Generic exception class; all objects thrown must be a Throwable

Table E.2 *Java.util Package*

Interfaces	
Enumeration	Methods for enumerating sets of values
Observer	Methods for enabling classes to be Observable objects
Classes	
BitSet	A set of bits
Date	The current system date, as well as methods for generating and parsing dates
Dictionary	An abstract class that maps between keys and values (superclass of HashTable)
Hashtable	A hash table
Observable	An abstract class for observable objects
Properties	A hash table that contains behaviour for setting and retrieving persistent properties of the system or a class
Random	Utilities for generating random numbers
Stack	A stack (a last-in-first-out queue)
StringTokenizer	Utilities for splitting strings into individual "token"
Vector	A growable array of Objects

Table E.3 *Java.io Package*

Interfaces	
taInput	Methods for reading machine-independent typed input streams
taOutput	Methods for writing machine-independent typed output streams

(Conta

FilenameFilter	Methods for filtering file names
Classes	
BufferedInputStream	A buffered input stream
BufferedOutputStream	A buffered output stream
ByteArrayInputStream	An input stream from a byte array
ByteArrayOutputStream	An output stream to a byte array
DataInputStream	Enables you to read primitive Java types (ints, chars, booleans, and so on) from a stream in a machine-independent way
DataOutputStream	Enables you to write primitive Java data types (ints, chars, booleans, and so on) to a stream in a machine-component way
File	Represents a file on the host's file system
FileDescriptor	Holds onto the UNIX-like file descriptor of a file or socket
FileInputStream	An input stream from a file, constructed using a filename of descriptor
FileOutputStream	An output stream to a file, constructed using a filename or descriptor
FilterInputStream	Abstract class which provides a filter for input streams (and for adding stream functionality such as buffering)
FilterOutputStream	Abstract class which provides a filter for output streams (and for adding stream functionality such as buffering)
Input Stream	An abstract class representing an input stream of bytes; the parent of all input streams in this package
LineNumberInputStream	An input stream that keeps track of line numbers
OutputStream	An abstract class representing an output stream of bytes; the parent of all output stream in this package
PipedInputStream	A piped input stream, which should be connected to a PipedOutputStream to be useful
PipedOutputStream	A piped output stream, which should be connected to a PipedInputStream to be useful (together they provide safe communication between threads)
PrintStream	An output stream for printing (used by System.out.println(...))
PushbackInputStream	An input stream with a 1-byte push back buffer
RandomAccessFile	Provides random access to a file, constructed from filenames, descriptors or objects
SequenceInputStream	Converts a sequence of input streams into a single input stream
StreamTokenizer	Converts an input stream into a series of individual tokens
StringBufferInputStream	An input stream from a String object

Table E.4 *Java.awt Package*

Interfaces	
LayoutManager	Methods for laying out containers
MenuContainer	Methods for menu-related containers

(Contd)

Classes	
BorderLayout	A layout manager for arranging items in border formation
Button	A UI pushbutton
Canvas	A canvas for drawing and performing other graphics operations
CardLayout	A layout manager for HyperCard-like metaphors
Checkbox	A checkbox
CheckboxGroup	A group of exclusive checkboxes (radio buttons)
CheckboxMenuItem	A toggle menu item
Choice	A popup menu of choices
Color	An abstract representation of a color
Component	The abstract generic class for all UI components
Container	Abstract behaviour for a component that can hold other components or containers
Dialog	A window for brief interactions with users
Dimension	An object representing width and height
Event	An object representing events caused by the system or based on user input
FileDialog	A dialog for getting filenames from the local file system
FlowLayout	A layout manager that lays out objects from left to right in rows
Font	An abstract representation of a font
FontMetrics	Abstract class for holding information about a specific font's character shapes and height and width information
Frame	A top-level window with a title
Graphics	Abstract behaviour for representing a graphics context, and for drawing and painting shapes and objects
GridBagConstraints	Constraints for components laid out using GridBagLayout
GridBagLayout	A layout manager that aligns components horizontally and vertically based on their values from GridBagConstraints
GridLayout	A layout manager with rows and columns; elements are added to each cell in the grid
Image	An abstract representation of a bitmap image
Insets	Distances from the outer border of the window; used to layout components
Label	A text label for UI components
List	A scrolling list
MediaTracker	A way to keep track of the status of media objects being loaded over the Net
Menu	A menu, which can contain menu items and is a container on a menubar
Menubar	A menubar (container for menus)
MenuComponent	The abstract superclass of all menu elements
MenuItem	An individual menu item
Panel	A container that is displayed
Point	An object representing a point (x and y coordinates)
Polygon	An object representing a set of points
Rectangle	An object representing a rectangle (x and y coordinates for the top corner, plus width and height)
Scrollbar	A UI scrollbar object

(Contd)

TextArea	A multilane, scrollable, editable text field
TextComponent	The supeclass of all editable text components
TextField	A fixed-size editable text field
Toolkit	Abstract behaviour for binding the abstract AWT classes to a platform-specific toolkit implementation
Window	A top-level window, and the superclass of the Frame and Dialog classes

Table E.5 *Java.awt.image Package*

Interfaces	
ImageConsumer	Methods for receiving image created by an ImageProducer
ImageObserver	Methods to track the loading and construction of an image
ImageProducer	Methods for producing image data received by an ImageConsumer
Classes	
ColorModel	An abstract class for managing color information for images
CropImageFilter	A filter for cropping images to a particular size
DirectColorModel	A specific color model for managing and translating pixel color values
FilteredImageSource	An ImageProducer that takes an image and an ImageFilter object, and produces an image for an ImageConsumer
ImageFilter	A filter that takes image data from an ImageProducer, modifies it in some way, and hands it off to an ImageConsumer
IndexColorModel	A specific color model for managing and translating color values in a fixed-color map
MemoryImageSource	An image producer that gets its image from memory; used after constructing an image by hand
PixelGrabber	An ImageConsumer that retrieves a subset of the pixels in an image
RGBImageFilter	Abstract behaviour for a filter that modifies the RGB values of pixels in RGB images

Table E.6 *Java.applet Package*

Interfaces	
AppletContext	Methods to refer to applet's context
AppletStub	Methods to implement applet viewers
AudioClip	Methods to play audio files
Classes	
Applet	The base applet class

Table E.7 *Java.net Package*

Classes	
`ContentHandler`	Abstract behaviour for reading data from a URL connection and constructing the appropriate local object, based on MIME types
`DatagramPacket`	A datagram packet (UDP)
`DatagramSocket`	A datagram socket
`InetAddress`	An object representation of an Internet host (host name, IP address)
`ServerSocket`	A sever-side socket
`Socket`	A socket
`SocketImpl`	An abstract class for specific socket implementations
`URL`	An object representation of a URL
`URLConnection`	Abstract behaviour for a socket that can handle various Web-based protocols (http, ftp, and so on)
`URLEncoder`	Turns strings into x-www-form-urlencoded format
`URLStreamHandler`	Abstract class for managing streams to object referenced by URLs

APPENDIX

I

Deprecated Classes and Methods

I.1 INTRODUCTION

As a part of the effort to enhance the performance and capabilities of the Java language, Sun Microsystems has altered and eliminated many classes and methods during upgradations. The altered methods have been added as new methods and the older ones have been retained in order to maintain backward compatibility with older versions of Java. However, the older methods have been marked "deprecated".

A deprecated method means that it has lost its importance and likely to be phased out of future Java versions. Although programs that use deprecated methods will still compile and work, the compiler will generate warning messages. It is recommended that programs be modified to eliminate the use of any deprecated methods and classes due to two reasons.

1. Modified programs will retain compatibility with future releases of Java.
2. Many new methods provide better implementations and therefore make programs faster and more efficient.

I.2 DEPRECATED CLASSES AND METHODS OF VERSION 1.0

A large number of classes and methods of version 1.0 have been declared deprecated. This appendix gives package-wise tables (Tables I.1 to I.4) that list class-wise methods that have been declared deprecated in Java 1.1. Tables also indicate alternative replacements. Table I.5 gives a list of version 1.0 classes that have been declared deprecated totally.