APPENDIX A: SUMMARY OF IMPORTANT JAVA FEATURES

A - 1: Literals

Literals	Explanation
`'A'` (means 65 or 0x41) `'0'` (means 48 or 0x30) `'1'` (means 49 or 0x31) `'a'` (means 97 or 0x61)	Character literals (i.e., literals of type **`char`**) are written within single quotation marks. A character literal like 'A' means the numerical value 65.
`'\n'` (newline, 0x0A) `'\b'` (backspace, 0x08) `'\r'` (carriage return, 0x0D) `'\\'` (backslash, 0x5C) `'\"'` (double quote, 0x22) `'\''` (single quote, 0x27) `'\t'` (tab, 0x09) `'\0'` (NULL, 0x00) `'\u0041'` (means 'A')	Special character literals are written by utilizing a so-called escape character, the backslash \. When a backslash precedes a symbol, the compiler realizes that the symbol denotes something other than the usual meaning of the symbol. With the prefix \u it is possible to give the hexadecimal Unicode character code of a character.
`123` (means 0x7B) `257` (means 0x101) `0x31` (means 49 or '1') `0x41` (means 65) `0xFFFF` (means 65535) `123L` (a **`long`** literal)	Integer literals can be written in different numbering systems. Prefix 0x (or alternatively 0X) identifies hexadecimal literals. The compiler recognizes numerical literals on the basis that they always begin with a numerical symbol. An integer literal like 123 can be assigned to all types of integral variables. The compiler issues an error message if a literal is too large to fit into a variable. The letter L at the end of an integer literal makes it a literal of type **`long`**. Java does not have binary literals.
`23.45` (means 2345e-2) `2.345` (means 2345e-3) `2.998e8` (means 299800000) `3.445e-2` (means 0.03445) `34.45e-3` (means 0.03445) `34.45e-3F` (**`float`** literal) `2.998e8F` (**`float`** literal)	Floating-point literals that can be stored in variables of type **`float`** and **`double`** can be expressed either in decimal or exponential (scientific) notation. The decimal point is symbolized by . (the full stop). The comma (,) is not used in floating-point literals. Floating-point literals of type **`float`** must have an F (or alternatively an f) at the end.
`"ABCDE"` (Length is 5) `"\nABCDE"` (Length is 6) `"\nABCDE."` (Length is 7) `"\n\"ABCDE.\""` (Length is 9) `"\n\n\n\n"` (Length is 4) `"\\ABCDE\\"` (Length is 7)	String literals are written with double quote characters. A string literal can be used to create an object of type **`String`**. Special characters can be included in string literals by using the same escape mechanism as is used in the case of character literals.
`false` `true`	Literals of type **`boolean`** are the two words **`true`** and **`false`**.
`null`	The keyword **`null`** means that no object is being referenced. This word can be assigned as a value to object references. **`null`** is the default value when object references are fields of a class or array elements.

A - 2: Variables, constants, and arrays of basic types

Declarations	Examples

Variable declarations
 	
The built-in variable types of Java are **byte**, **char**, **short**, **int**, **long**, **float**, **double**, and **boolean**. The storage capacities of different built-in variable types are shown in Table 5-1.
	`char  character_from_keyboard ;`
	`short  given_small_integer ;`
	`int  integer_from_keyboard ;`
	`long  multiplication_result ;`
	
	When variables are used as local variables inside methods, they must be assigned values before they can be used. When variables are used as fields of classes, they are automatically assigned zero values. (**boolean** fields and fields that are object references are automatically assigned values **false** and **null**, respectively.)

Initialized variables	`char  user_selection  =  '?' ;`
	`byte  mask_for_most_significant_bit  =  (byte) 0x80 ;`
	`int  character_index  =  0 ;`
	`int  bit_mask  =  0x80000000 ;`
	`long  speed_of_light  =  299793000L ;`
	`float  kilometers_to_miles  = 1.6093F ;`
	`double  value_of_pi  =  3.14159 ;`
	`boolean  text_has_been_modified  =  false ;`

Constant declarations
 	
Constants are "variables" whose values cannot be changed.
	`final int  LENGTH_OF_NORMAL_YEAR  =  365 ;`
	`final int  LENGTH_OF_LEAP_YEAR  =  366 ;`
	
	`final double LENGTH_OF_YEAR_IN_SECONDS  =  31558149.5 ;`
	`final float EXACT_LENGTH_OF_YEAR_IN_DAYS  =  365.256F ;`

Array declarations and creations	`char[] array_of_characters ;`
	`array_of_characters  =  new  char[ 50 ] ;`
	`int[]  array_of_integers  =  new  int[ 60 ] ;`
	`int[]  integers_to_power_of_two =`
	`{ 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121 } ;`
	`int[]  two_to_power_of_integer =`
	`{ 1, 2, 4, 8, 16, 0x20, 0x40, 0x80, 0x100 } ;`
	`char[]  hexadecimal_digits  =`
	`{ '0', '1', '2', '3', '4', '5', '6', '7',`
	`'8', '9', 'A', 'B', 'C', 'D', 'E', 'F' } ;`
	`double[] pi_times_integer  =`
	`{ 0, 3.1416, 6.2832, 9.4248, 12.5664 } ;`
	`float[] centimeters_to_inches  =`
	`{ 0, 0.3937F, 0.7874F, 1.1811F, 1.5748F, 1.9685F};`
	
	`int[][]  some_two_dimensional_array  =  new int[ 5 ][ 9 ] ;`
	
	If an array is created so that it is not initialized with values listed inside braces, array elements are automatically initialized with zeroes. The elements of an array of type **boolean[]** are automatically initialized with **false**. Arrays containing object references are initialized with **null**.

A - 3: String objects, other objects, and arrays of objects

Declaration	Examples

String declarations and creations	`String  some_string  ;  // declares a string reference`
	`String  another_string  =  ""  ;  // an empty string`
	`String  third_string  =  "text inside string object" ;`
	
	`char[]  some_letters  =  { 'K', 'a', 'r', 'i' } ;`
	`String  some_name  =  new  String( some_letters ) ;`
	`String  some_copied_string  =  new String( some_name ) ;`

Object references and creations	`ClassName  object_name ;  // declares an object reference`
	`object_name  =  new  ClassName( ... ) ; // object creation`
	
	`Date first_day_of_this_millennium = new Date( 1, 1, 2000 ) ;`
	`Date last_day_of_this_millennium = new Date("12/31/2999") ;`
	
	`Object  anything ;  // This can reference any object`

Arrays of objects	An array of objects is actually an array of object references. Right after its creation, the elements of an array of objects contain **null** references.
	
	`String[]  any_array_of_strings ;`
	
	`String[]  array_of_strings  =  new  String[ 9 ] ;`
	`array_of_strings[ 0 ]  =  "some text line" ;`
	`array_of_strings[ 1 ]  =  "another text line" ;`
	`...`
	
	`// The following is an initialized array of strings`
	`String[]  largest_moons_of_jupiter  =`
	`{ "Io", "Ganymede", "Europa", "Callisto" } ;`
	
	`Date[]  days_of_this_millennium  =  new  Date[ 365243 ] ;`
	`days_of_this_millennium[ 0 ]  =  new Date( 1, 1, 2000 ) ;`
	`days_of_this_millennium[ 1 ]  =  new Date( 2, 1, 2000 ) ;`
	`days_of_this_millennium[ 2 ]  =  new Date( 3, 1, 2000 ) ;`
	`...`
	
	`SomeClass[]  array_of_objects  =  new  SomeClass[ 10 ] ;`
	`array_of_objects[ 0 ]  =  new  SomeClass( ... ) ;`
	`array_of_objects[ 1 ]  =  new  SomeClass( ... ) ;`
	`...`

A - 4: Expressions

The word "expression" is an important term when speaking about the grammars of programming languages. The following are examples of valid Java expressions:

```
1
254
true
some_variable
some_variable + 3
( some_variable * another_variable )
( first_variable + second_variable ) / third_variable
some_array[ 3 ]
array_of_objects[ object_index ]
some_string.length()
some_object.some_method()
some_object.SOME_STATIC_CONSTANT
```

You can see that literals, references to variables, mathematical calculations, references to objects in arrays, method calls, etc. are all expressions in Java. Expressions are parts of larger program constructs such as assignment statements, **`if`** constructs and loops. Expressions obtain some values when a program is being executed. When an expression represents a mathematical operation, we can say that it is a mathematical or arithmetic expression. Expressions that get the values **`true`** or **`false`** are boolean expressions.

By using the term expression it is easy to speak, for example, about the operators of a programming language. The use of the addition operator (+) can be specified

expression **`+`** *expression*

which can mean, for example, all the following expressions

```
some_variable + 254
some_variable + another_variable
some_variable + some_string.length()
33 + array_of_integers[ integer_index ]
```

A - 5: Assignments and left-side expressions

When you put the assignment statement

```
1 = 1 ;
```

in a program, the compiler considers it as an error and says something like "unexpected type; required: variable; found: value" The above statement tries to assign a value to a literal, and that is not possible. I use the term "left-side expressions" to refer to expressions that are allowed on the left side of an assignment operation. A literal or a method call are not left-side expressions. Typical left-side expressions are variables, object references, references to public fields of objects, and indexed positions of arrays. The following kinds of assignment statements are thus possible

```
some_variable  =  ...
some_object  =  ...
some_object.some_public_field  =  ...
array_of_integers  =  ...
array_of_integers[ integer_index ]  =  ...
array_of_integers[ integer_index + 1 ]  =  ...
array_of_objects[ object_index ]  =  ...
```

A - 6: The most important Java operators in order of precedence

Symbol	Operator name	Notation	Comments

`.`
`[]`
`()`
	member selection
	array indexing
	method call
		`object_name.member_name`
		`array_name[` *expression* `]`
		`method_name(` *list of expressions* `)`
			All three operators mentioned here have the same, the highest, precedence.

`++`
`--`
`~`
`!`
`-`
`+`
`( Type )`
`new`
	increment
	decrement
	complement
	not
	unary minus
	unary plus
	type cast
	object creation
		left-side-expression `++`
		left-side-expression `--`
		`~` *expression*
		`!` *expression*
		`-` *expression*
		`+` *expression*
		`( Type )` *expression*
		`new Type(` *list of expressions* `)`
			These unary operators are right-to-left assco-ciative. All other operators, excluding the assignment opera-tors, are left-to-right associative.

`*`
`/`
`%`
	multiplication
	division
	remainder
		expression `*` *expression*
		expression `/` *expression*
		expression `%` *expression*
			Arithmetic operators (multiplicative).

`+` `-`	addition subtraction	*expression* `+` *expression* *expression* `-` *expression*	Arithmetic operators (additive).

`<<`
`>>`
`>>>`
	shift left
	shift right
	shift right (zero fill)
		expression `<<` *expression*
		expression `>>` *expression*
		expression `>>>` *expression*
			Bitwise shift opera-tors.

`<`
`<=`
`>`
`>=`
`instanceof`
	less than
	less than or equal
	greater than
	greater than or equal
	type compatibility
		expression `<` *expression*
		expression `<=` *expression*
		expression `>` *expression*
		expression `>=` *expression*
		expression `instanceof Type`
			Relational operators.

`==` `!=`	equal not equal	*expression* `==` *expression* *expression* `!=` *expression*	Relational operators or equality operators.
`&`	bitwise AND	*expression* `&` *expression*	
`^`	bitwise exclusive OR	*expression* `^` *expression*	
|	bitwise OR	*expression* | *expression*	
`&&`	(conditional) logical AND	*expression* `&&` *expression*	
||	(conditional) logical OR	*expression* || *expression*	

`=`
`+=`
`-=`
`*=`
`etc.`
	basic assignment
	add and assign[a]
	subtract and assign
	multiply and assign
	etc.
		left-side-expression `=` *expression*
		left-side-expression `+=` *expression*
		left-side-expression `-=` *expression*
		left-side-expression `*=` *expression*
		etc.
			Assignment opera-tors are right-to-left associative. All arith-metic operators and most bit operators can be combined with the assignment operator =.

a. Operators `+=`, `-=`, `*=`, etc. work so that
`some_variable += 3 ;` means the same as
`some_variable = some_variable + 3 ;` and
`some_variable *= another variable ;` means the same as
`some_variable = some_variable * another_variable ;`

A - 7: Control structures to make decisions (selections)

Control structure	Description
Simple **if** construct	**if (** *boolean expression* **)** **{** One or more statements that will be executed if the boolean expression, given in parentheses above, is true. These statements will not be executed at all if the boolean expression is false (i.e. not true). **}**
if-else construct	**if (** *boolean expression* **)** **{** One or more statements that will be executed if the boolean expression, given in parentheses above, is true. **}** **else** **{** One or more statements that will be executed if the boolean expression, given in parentheses above, is false (i.e. not true). **}**
if-else if ... construct	**if (** *boolean expression 1* **)** **{** One or more statements that will be executed if and only if boolean expression 1 is true. **}** **else if (** *boolean expression 2* **)** **{** One or more statements that will be executed if and only if boolean expression 2 is true and boolean expression 1 is false. **}** **else** **{** One or more statements that will be executed if and only if neither boolean expression 1 nor boolean expression 2 is true. **}**
switch-case construct	**switch (** *arithmetic expression* **)** **{** **case** v_1**:** Statements which will be executed if the arithmetic expression has value v_1 **break ;** **case** v_2**:** Statements which will be executed if the arithmetic expression has value v_2 **break ;** **case** v_n**:** Statements to be executed when the arithmetic expression has value v_n **break ;** **default:** Statements which will be executed if none of the cases matched the value of the arithmetic expression **break ;** **}**

A - 8: Control structures to perform repetitions (iterations)

Control structure	Description
`while` loop	**`while (`** *boolean expression* **`)`** **`{`** One or more internal statements that will be repeatedly executed as long as the boolean expression, given in parentheses above, is true. **`}`**
`do-while` loop	**`do`** **`{`** One or more statements that will be first executed once, and then repeatedly executed as long as the boolean expression, given below in parentheses, is true. **`}`** **`while (`** *boolean expression* **`) ;`**
`for` loop	**`for (`** *assignment statement* **`;`** *boolean expression* **`;`** *increment or decrement statement* **`)`** **`{`** One or more internal statements that will be repeatedly executed as long as the boolean expression given above is true. When the boolean expression becomes false, the statements that follow this **`for`** loop will be executed. **`}`** An index variable may be declared in a **`for`** loop in the following way **`for ( int some_index = 0 ;`** **`...`** The scope of this kind of variable is within the internal statements of the loop.
"foreach" loop	**`for ( Type object_name : collection_name )`** **`{`** One or more statements that will be executed for each object in the collection. **`object_name`** refers to the object currently being processed, and the loop automatically processes all objects of the collection. The collection being processed can be a conventional array, an **`ArrayList`** array, or some other kind of collection that implements the **`Iterable`** interface. **`}`**

A - 9: Some basic Java method structures

Method type	Example

A static method named `main()` is the method that is invoked by the Java virtual machine when an executable program is run on a computer. It is mandatory to declare a formal parameter for the `main()` method. In this book, the name of the parameter is `not_in_use` when it is not used.	`public static void main( String[] not_in_use )`
	`{`
	`  ...`
	`}`

The parameter that is supplied by the operating system and the virtual machine to method `main()` is an array of strings that contains the data that is supplied from the command line. In this book, the parameter is named `command_line_parameters` when it is used by the `main()` method.	`public static void main(`
	`  String[] command_line_parameters )`
	`{`
	`  ...`
	`}`

A method that neither takes parameters nor outputs a return value.	`void method_name()`
	`{`
	`  ...`
	`}`

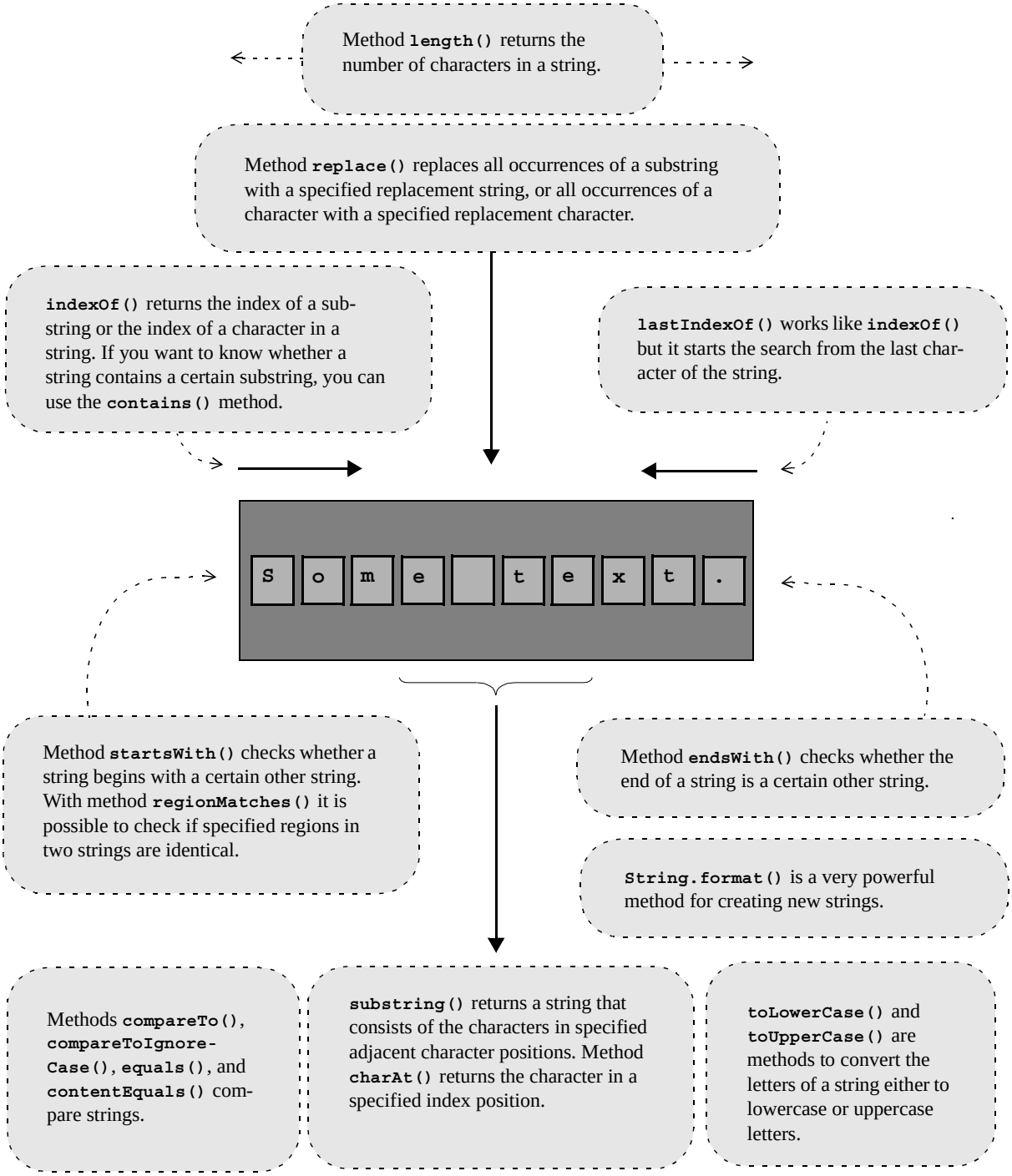
A method to which two parameters of type `int` can be passed by value.	`void method_name( int first_parameter,`
	`  int second_parameter )`
	`{`
	`  ...`
	`}`

A method that takes two `int` values as input parameters and returns an `int` value with a `return` statement.	`int method_name( int first_parameter,`
	`  int second_parameter )`
	`{`
	`  int value_to_caller ;`
	`  ...`
	`  return value_to_caller ;`
	`}`

A method that takes an array of type `int[]` as a parameter. When arrays and other objects are passed as parameters, an array reference or an object reference is passed as a value to the called method. Thus the called method and the caller can access the same array or the same object.	`void method_name( int[] array_of_integers )`
	`{`
	`  ...`
	`}`

A - 10: String methods

The drawing on this page explains briefly many of the string methods. To find a more accurate description of the methods, please go to page 220.



A - 11: Mechanisms for keyboard input and screen output

The mechanisms to output data to the screen and read data from the keyboard are explained at the end of Chapter 5.

A - 12: Input/output from/to files

Activity	How to make it happen?
	To perform file operations in Java, the package **java.io** must be imported.
Open a text file for input	`BufferedReader input_file = new BufferedReader( new FileReader( "filename.txt" ) ) ;`
Open a text file for output	`PrintWriter output_file = new PrintWriter( new FileWriter( "filename.txt" ) ) ;` `PrintWriter growing_text_file = new PrintWriter( new FileWriter( "append_here.txt", true ) ) ;`
Check if file opened successfully	An exception is thrown if file opening does not succeed. File operations must be carried out by using a `try-catch(-finally)` construct.
Output text to text file	`output_file.println( "This line goes to file" ) ;`
Input text from text file	`String text_line_from_file = input_file.readLine() ;` `readLine()` returns a `null` when the end of file has been encountered.
Open a file in binary form for reading	`FileInputStream binary_input_file = new FileInputStream( "important.data" );`
Open a file in binary form for writing	`FileOutputStream binary_output_file = new FileOutputStream("important.data" ) ;` `FileOutputStream growing_binary_file = new FileOutputStream( "important.data", true ) ;`
Read bytes from a binary file	`int number_of_bytes_actually_read = binary_input_file.read( array_of_bytes, array_position, // 0, 1, 2, ... desired_number_of_bytes ) ;` `int number_of_bytes_actually_read = binary_input_file.read( array_of_bytes ) ;`
Write bytes to a binary file	`binary_output_file.write( array_of_bytes, array_position, // 0, 1, 2, 3, ... number_of_bytes_to_write ) ;` `binary_output_file.write( array_of_bytes ) ;`

Close an open file	`input_file.close() ;`
	`output_file.close() ;`
	`binary_input_file.close() ;`
	`binary_output_file.close() ;`

A - 13: Data conversions

Conversion mechanism	How to use it?

Parsing methods	Standard Java wrapper classes (e.g. **Short**, **Integer**, **Long**, **Byte**, **Float**, and **Double**) provide static methods like **parseShort()**, **parseInt()**, **parseLong()**, etc., which can be used to parse a character string so that the string is converted to a numerical type. A string can be converted to a **double** value in the following way
	`String value_of_pi_as_string = "3.14159" ;`
	`double value_of_pi =`
	`Double.parseDouble( value_of_pi_as_string ) ;`
	The parsing methods are useful, for example, when we want to convert a string that contains a binary or a hexadecimal value. The statement
	`System.out.print( "\n " + Integer.parseInt( "123" )`
	`+ " " + Integer.parseInt( "1111011", 2 )`
	`+ " " + Integer.parseInt( "7B", 16 ) ) ;`
	would print
	`123 123 123`

toString() methods	All Java classes have a method named **toString()** that can convert an object to a string. A **toString()** method can be invoked for an object by calling it explicitly or by using the string concatenation operator (+). The statement
	`System.out.print( "" + some_object ) ;`
	would invoke a **toString()** method for the object referenced by **some_object**, and print the string to the screen. Some of the above-mentioned standard wrapper classes provide static methods like **toBinaryString()** and **toHexString()** with which it is possible to convert numerical values to strings in which the numbering system is not the decimal system. For example, the statements
	`String hexadecimal_string = Integer.toHexString( 33 );`
	`System.out.print( hexadecimal_string ) ;`
	would print 21 to the screen.

String.format() method	The static **String.format()** method is a very powerful tool to convert numerical values to strings. You have to use format specifiers like **%d**, **%X**, **%f**, etc., to make the method perform the desired conversions. The conversion shown above can alternatively be carried out with the statement `String hexadecimal_string = String.format( "%X", 33 );`

valueOf() methods	The standard class **String** and the above-mentioned wrapper classes provide many static **valueOf()** methods. For example, in the statement
	`double value_of_pi =`
	`Double.valueOf( value_of_pi_as_string ) ;`
	a string is first converted to a **Double** object and then unboxing takes place.

Casting operations	Casting is a mechanism to temporarily convert a data item to another type. Casting is usually used inside a larger statement. For example, the following statement converts a value of type **char** to an **int** value before printing: `System.out.print( (int) some_character ) ;` The above statement prints the character code of a character, not the character. Casting is required, for example, in assignment statements in which the value of a large variable is stored in a small variable, e.g., when the value of a **long** variable is copied to a variable of type **int**.

A - 14: Java class declaration

There are many different possibilities to declare classes in Java. Actually, all of Part III of this book is a long discussion of the nature of Java classes. A class declaration is identified with the reserved keyword **`class`**. If keyword **`public`** precedes the **`class`** keyword, the class is visible outside its package. A package is a collection that can contain many classes. Keyword **`abstract`** must be written before the **`class`** keyword if the class contains one or more abstract methods. If the **`final`** keyword precedes the **`class`** keyword, it is not possible to derive new classes from the class.

Keyword **`extends`** specifies that another class is inherited. Keyword **`implements`** specifies that one or more interfaces are implemented. A class can inherit from one superclass. It can implement one or more interfaces.

```
class ClassName  extends  SuperclassName
                 implements  SomeInterfaceName, SomeOtherInterfaceName
{
   protected int some_field ;
   ...

   public ClassName()
   {
      ...
   }

   public int get_some_field()
   {
      return  some_field ;
   }

   public void some_method( int some_parameter )
   {
      ...
   }

   public void some_other_method( ... )
   {
      ...
   }

   public String toString()
   {
      ...
      return object_as_string ;
   }
}
```

Usually classes have several constructors. A constructor has the same name as the class. A constructor is called automatically when an object (instance) of a class is created. A default constructor is one that can be called without giving any parameters.

An accessor method is one that is used to either read or write a field of a class.

All classes have a method named **`toString()`** because such a method is declared in class **`Object`** that is the superclass of all Java classes. If a class declaration does not contain a **`toString()`** method, it is inherited from class **`Object`** or from some other class in a class hierarchy.

All non-**`static`** and non-**`private`** methods of Java classes can be polymorphic methods that are overridden in lower classes. When a polymorphic method is called for an object, the correct version of the method that corresponds with the object's type is automatically selected. If you want to prevent the overriding of a method, you can declare it with the **`final`** keyword.