Language Fundamentals

2

Programmer I Exam Objectives	
[1.1] Define the scope of variables ❍ *See also §4.4, p. 114.*	*§2.4, p. 44*
[2.1] Declare and initialize variables (including casting of primitive data types) ❍ *For casting of primitive data types, see §5.6, p. 160.*	*§2.3, p. 40* *§2.4, p. 42*
[2.2] Differentiate between object reference variables and primitive variables	*§2.3, p. 40*
Supplementary Objectives	
• Be able to identify the basic elements of the Java programming language: keywords, identifiers, literals, and primitive data types	*§2.1, p. 28* *§2.2, p. 37*

2.1 Basic Language Elements

Like any other programming language, the Java programming language is defined by *grammar rules* that specify how *syntactically* legal constructs can be formed using the language elements, and by a *semantic definition* that specifies the *meaning* of syntactically legal constructs.

Lexical Tokens

The low-level language elements are called *lexical tokens* (or just *tokens*) and are the building blocks for more complex constructs. Identifiers, numbers, operators, and special characters are all examples of tokens that can be used to build high-level constructs like expressions, statements, methods, and classes.

Identifiers

A name in a program is called an *identifier*. Identifiers can be used to denote classes, methods, variables, and labels.

In Java, an *identifier* is composed of a sequence of characters, where each character can be either a *letter* or a *digit*. However, the first character in an identifier must always be a letter, as explained later.

Since Java programs are written in the Unicode character set (p. 32), characters allowed in identifier names are interpreted according to this character set. Use of the Unicode character set opens up the possibility of writing identifier names in many writing scripts used around the world. As one would expect, the characters `A-Z` and `a-z` are letters, and characters from `0-9` are digits. A *connecting punctuation character* (such as *underscore* _) and any *currency symbol* (such as `$`, `¢`, `¥`, or `£`) are also allowed as letters in identifier names, but these characters should be used judiciously.

Identifiers in Java are *case sensitive*. For example, `price` and `Price` are two different identifiers.

Examples of Legal Identifiers

```
number, Number, sum_$, bingo, $$_100, _007, mål, grüß
```

Examples of Illegal Identifiers

```
48chevy, all@hands, grand-sum
```

The name `48chevy` is not a legal identifier because it starts with a digit. The character `@` is not a legal character in an identifier. It is also not a legal operator, so that `all@hands` cannot be interpreted as a legal expression with two operands. The character `-` is not a legal character in an identifier, but it is a legal operator; thus `grand-sum` could be interpreted as a legal expression with two operands.

Keywords

Keywords are reserved words that are predefined in the language and cannot be used to denote other entities. All Java keywords are lowercase, and incorrect usage results in compile-time errors.

Keywords currently defined in the language are listed in Table 2.1. In addition, three identifiers are reserved as predefined *literals* in the language: the `null` reference, and the boolean literals `true` and `false` (Table 2.2). Keywords currently reserved, but not in use, are listed in Table 2.3. A reserved word cannot be used as an identifier. The index contains references to relevant sections where currently used keywords are explained.

Table 2.1 *Keywords in Java*

abstract	default	if	private	this
assert	do	implements	protected	throw
boolean	double	import	public	throws
break	else	instanceof	return	transient
byte	enum	int	short	try
case	extends	interface	static	void
catch	final	long	strictfp	volatile
char	finally	native	super	while
class	float	new	switch	
continue	for	package	synchronized	

Table 2.2 *Reserved Literals in Java*

null	true	false

Table 2.3 *Reserved Keywords Not Currently in Use*

const	goto

Separators

Separators (also known as *punctuators*) are tokens that have meaning depending on the context in which they are used; they aid the compiler in performing syntax and semantic analysis of a program (Table 2.4). Depending on the context, brackets (`[]`), parentheses (`()`), and the dot operator (`.`) can also be interpreted as *operators* (§5.3, p. 150). See the index entries for these separators for more details.

Table 2.4 *Separators in Java*

{	}	[	]	(	)
.	;	,	...	@	::

Literals

A *literal* denotes a constant value; in other words, the value that a literal represents remains unchanged in the program. Literals represent numerical (integer or floating-point), character, boolean, or string values. In addition, the literal `null` represents the null reference. Table 2.5 shows examples of literals in Java.

Table 2.5 *Examples of Literals*

Integer	`2000`	`0`	`-7`		
Floating-point	`3.14`	`-3.14`	`.5`	`0.5`	
Character	`'a'`	`'A'`	`'0'`	`':'`	`'-'` `')'`
Boolean	`true`	`false`			
String	`"abba"`	`"3.14"`	`"for"`	`"a piece of the action"`	

Integer Literals

Integer data types comprise the following primitive data types: `int`, `long`, `byte`, and `short` (§2.2, p. 37).

The default data type of an integer literal is always `int`, but it can be specified as `long` by appending the suffix `L` (or `l`) to the integer value. The suffix `L` is often preferred because the suffix `l` and the digit `1` can be hard to distinguish. Without the suffix, the `long` literals `2000L` and `0L` will be interpreted as `int` literals. There is no direct way to specify a `short` or a `byte` literal.

In addition to the decimal number system, integer literals can be specified in the binary (*base* 2, *digits* `0`-`1`), octal (*base* 8, *digits* `0`-`7`), and hexadecimal (*base* 16, *digits* `0`-`9` and `a`-`f`) number systems. The digits `a` to `f` in the hexadecimal system correspond to decimal values `10` to `15`. Binary, octal, and hexadecimal numbers are specified with `0b` (or `0B`), `0`, and `0x` (or `0X`) as the base or radix prefix, respectively. Examples of decimal, binary, octal, and hexadecimal literals are shown in Table 2.6. Note that the leading `0` (zero) digit is not the uppercase letter `O`. The hexadecimal digits from `a` to `f` can also be specified with the corresponding uppercase forms (`A` to `F`). Negative integers (e.g., `-90`) can be specified by prefixing the minus sign (`-`) to the magnitude of the integer regardless of the number system (e.g., `-0b1011010`, `-0132`, or `-0X5A`). Integer representation is discussed in §5.5, p. 154.

Table 2.6 *Examples of Decimal, Binary, Octal, and Hexadecimal Literals*

Decimal	Binary	Octal	Hexadecimal
8	0b1000	010	0x8
10L	0b1010L	012L	0xaL
16	0b10000	020	0x10
27	0b11011	033	0x1b

Table 2.6 *Examples of Decimal, Binary, Octal, and Hexadecimal Literals (Continued)*

Decimal	Binary	Octal	Hexadecimal
90L	0b1011010L	0132L	0x5aL
-90	-0b1011010 or 0b11111111111111111111111110100110	-0132 or 037777777646	-0x5a or 0xffffffa6
-1	-0b1 or 0b11111111111111111111111111111111	-01 or 037777777777	-0x1 or 0xffffffff
2147483647 (i.e., $2^{31}-1$)	0b01111111111111111111111111111111	017777777777	0x7fffffff
-2147483648 (i.e., -2^{31})	0b10000000000000000000000000000000	020000000000	0x80000000
1125899906842624L (i.e., 2^{50})	0b10000000000000000000000000000000000000000000000000000L	040000000000000000L	0x4000000000000L

Floating-Point Literals

Floating-point data types come in two flavors: `float` or `double`.

The default data type of a floating-point literal is `double`, but it can be explicitly designated by appending the suffix `D` (or `d`) to the value. A floating-point literal can also be specified to be a `float` by appending the suffix `F` (or `f`).

Floating-point literals can also be specified in scientific notation, where `E` (or `e`) stands for *exponent*. For example, the `double` literal `194.9E-2` in scientific notation is interpreted as 194.9×10^{-2} (i.e., 1.949).

Examples of `double` *Literals*

```
0.0       0.0d       0D
0.49      .49        .49D
49.0      49.        49D
4.9E+1    4.9E+1D    4.9e1d   4900e-2  .49E2
```

Examples of `float` *Literals*

```
0.0F      0f
0.49F     .49F
49.0F     49.F       49F
4.9E+1F   4900e-2f   .49E2F
```

Note that the decimal point and the exponent are optional, and that at least one digit must be specified. Also, for the examples of `float` literals presented here, the suffix `F` is mandatory; if it was omitted, they would be interpreted as `double` literals.

Underscores in Numerical Literals

The underscore character (_) can be used to improve the readability of numerical literals in the source code. Any number of underscores can be inserted *between the digits* that make up the numerical literal. This rules out underscores adjacent to the sign (+, -), the radix prefix (0b, 0B, 0x, 0X), the decimal point (.), the exponent (e, E), and the data type suffix (l, L, d, D, f, F), as well as before the first digit and after the last digit. Note that octal radix prefix 0 is part of the definition of an octal literal and is therefore considered the first digit of an octal literal.

Underscores in identifiers are treated as letters. For example, the names _XXL and _XXL_ are two distinct legal identifiers. In contrast, underscores are used as a notational convenience for numerical literals, being ignored by the compiler when used in such literals. In other words, a numerical literal can be specified in the source code using underscores between digits, such that 2_0_1_5 and 20__15 represent the same numerical literal 2015 in source code.

Examples of Legal Use of Underscores in Numerical Literals

```
0b0111_1111_1111_1111_1111_1111_1111_1111
0_377_777_777            0xff_ff_ff_ff
-123_456.00              1_2.345_678e1_2
2009__08__13             49_03_01d
```

Examples of Illegal Use of Underscores in Numerical Literals

```
_0_b_01111111111111111111111111111111_
_0377777777_             _0_x_ffffffff_
+_123456_._00_           _12_._345678_e_12_
_20090813_               _490301_d_
```

Boolean Literals

The primitive data type boolean represents the truth values *true* and *false* that are denoted by the reserved literals true and false, respectively.

Character Literals

A character literal is quoted in single quotes ('). All character literals have the primitive data type char.

A character literal is represented according to the 16-bit Unicode character set, which subsumes the 8-bit ISO-Latin-1 and the 7-bit ASCII characters. In Table 2.7, note that digits (0 to 9), uppercase letters (A to Z), and lowercase letters (a to z) have contiguous Unicode values. A Unicode character can always be specified as a four-digit hexadecimal number (i.e., 16 bits) with the prefix \u.

Table 2.7 *Examples of Character Literals*

Character literal	Character literal using Unicode value	Character
`' '`	`'\u0020'`	Space
`'0'`	`'\u0030'`	0
`'1'`	`'\u0031'`	1
`'9'`	`'\u0039'`	9
`'A'`	`'\u0041'`	A
`'B'`	`'\u0042'`	B
`'Z'`	`'\u005a'`	Z
`'a'`	`'\u0061'`	a
`'b'`	`'\u0062'`	b
`'z'`	`'\u007a'`	z
`'Ñ'`	`'\u0084'`	Ñ
`'å'`	`'\u008c'`	å
`'ß'`	`'\u00a7'`	ß

Escape Sequences

Certain *escape sequences* define special characters, as shown in Table 2.8. These escape sequences can be single-quoted to define character literals. For example, the character literals `\t` and `\u0009` are equivalent. However, the character literals `\u000a` and `\u000d` should not be used to represent newline and carriage return in the source code. These values are interpreted as line-terminator characters by the compiler, and will cause compile-time errors. You should use the escape sequences `\n` and `\r`, respectively, for correct interpretation of these characters in the source code.

Table 2.8 *Escape Sequences*

Escape sequence	Unicode value	Character
`\b`	`\u0008`	Backspace (BS)
`\t`	`\u0009`	Horizontal tab (HT or TAB)
`\n`	`\u000a`	Linefeed (LF), also known as newline (NL)
`\f`	`\u000c`	Form feed (FF)
`\r`	`\u000d`	Carriage return (CR)
`\'`	`\u0027`	Apostrophe-quote, also known as single quote
`\"`	`\u0022`	Quotation mark, also known as double quote
`\\`	`\u005c`	Backslash

We can also use the escape sequence `\ddd` to specify a character literal as an octal value, where each digit `d` can be any octal digit (0–7), as shown in Table 2.9. The number of digits must be three or fewer, and the octal value cannot exceed `\377`; in other words, only the first 256 characters can be specified with this notation.

Table 2.9 *Examples of Escape Sequence* `\ddd`

Escape sequence `\ddd`	**Character literal**
`'\141'`	`'a'`
`'\46'`	`'&'`
`'\60'`	`'0'`

String Literals

A *string literal* is a sequence of characters that must be enclosed in double quotes and must occur on a single line. All string literals are objects of the class `String` (§8.4, p. 357).

Escape sequences as well as Unicode values can appear in string literals:

```
"Here comes a tab.\t And here comes another one\u0009!"          (1)
"What's on the menu?"                                            (2)
"\"String literals are double-quoted.\""                         (3)
"Left!\nRight!"                                                  (4)
"Don't split                                                     (5)
me up!"
```

In (1), the tab character is specified using the escape sequence and the Unicode value, respectively. In (2), the single apostrophe need not be escaped in strings, but it would be if specified as a character literal (`'\''`). In (3), the double quotes in the string must be escaped. In (4), we use the escape sequence `\n` to insert a newline. The expression in (5) generates a compile-time error, as the string literal is split over several lines. Printing the strings from (1) to (4) will give the following result:

```
Here comes a tab.    And here comes another one    !
What's on the menu?
"String literals are double-quoted."
Left!
Right!
```

One should also use the escape sequences `\n` and `\r`, respectively, for correct interpretation of the characters `\u000a` (newline) and `\u000d` (form feed) in string literals.

Whitespace

A *whitespace* is a sequence of spaces, tabs, form feeds, and line terminator characters in a Java source file. Line terminators include the newline, carriage return, or a carriage return–newline sequence.

A Java program is a free-format sequence of characters that is *tokenized* by the compiler—that is, broken into a stream of tokens for further analysis. Separators and operators help to distinguish tokens, but sometimes whitespace has to be inserted explicitly as a separator. For example, the identifier `classRoom` will be interpreted as a single token, unless whitespace is inserted to distinguish the keyword `class` from the identifier `Room`.

Whitespace aids not only in separating tokens, but also in formatting the program so that it is easy to read. The compiler ignores the whitespace once the tokens are identified.

Comments

A program can be documented by inserting comments at relevant places in the source code. These comments are for documentation purposes only and are ignored by the compiler.

Java provides three types of comments that can be used to document a program:

- A single-line comment: `// ... to the end of the line`
- A multiple-line comment: `/* ... */`
- A documentation (Javadoc) comment: `/** ... */'`

Single-Line Comment

All characters after the comment-start sequence `//` through to the end of the line constitute a *single-line comment*.

```
// This comment ends at the end of this line.
int age;       // From comment-start sequence to the end of the line is a comment.
```

Multiple-Line Comment

A *multiple-line comment*, as the name suggests, can span several lines. Such a comment starts with the sequence `/*` and ends with the sequence `*/`.

```
/* A comment
   on several
   lines.
*/
```

The comment-start sequences (`//`, `/*`, `/**`) are not treated differently from other characters when occurring within comments, so they are ignored. This means that trying to nest multiple-line comments will result in a compile-time error:

```
/* Formula for alchemy.
   gold = wizard.makeGold(stone);
   /* But it only works on Sundays. */
*/
```

The second occurrence of the comment-start sequence `/*` is ignored. The last occurrence of the sequence `*/` in the code is now unmatched, resulting in a syntax error.

Documentation Comment

A *documentation comment* is a special-purpose multiple-line comment that is used by the `javadoc` tool to generate HTML documentation for the program. Documentation comments are usually placed in front of classes, interfaces, methods, and field definitions. Special tags can be used inside a documentation comment to provide more specific information. Such a comment starts with the sequence `/**` and ends with the sequence `*/`:

```
/**
 *  This class implements a gizmo.
 *  @author K.A.M.
 *  @version 4.0
 */
```

For details on the `javadoc` tool, see the tools documentation provided by the JDK.

Review Questions

2.1 Which of the following is not a legal identifier?

Select the one correct answer.

(a) `a2z`
(b) `ödipus`
(c) `52pickup`
(d) `_class`
(e) `ca$h`
(f) `_8to5`

2.2 Which of the following are not legal literals in Java?

Select the four correct answers.

(a) `0Xbad`
(b) `0B_101_101`
(c) `09`
(d) `+_825`
(e) `1_2e4f`
(f) `'\x'`
(g) `"what\'s your fancy?"`

2.3 Which statement is true?

Select the one correct answer.

(a) `new` and `delete` are keywords in the Java language.
(b) `try`, `catch`, and `thrown` are keywords in the Java language.
(c) `static`, `unsigned`, and `long` are keywords in the Java language.
(d) `exit`, `class`, and `while` are keywords in the Java language.
(e) `return`, `goto`, and `default` are keywords in the Java language.
(f) `for`, `while`, and `next` are keywords in the Java language.

2.4 Which of the following is not a legal comment in Java?

Select the one correct answer.

(a) `/* // */`
(b) `/* */ //`
(c) `// /* */`
(d) `/* /* */`
(e) `/* /* */ */`
(f) `// //`

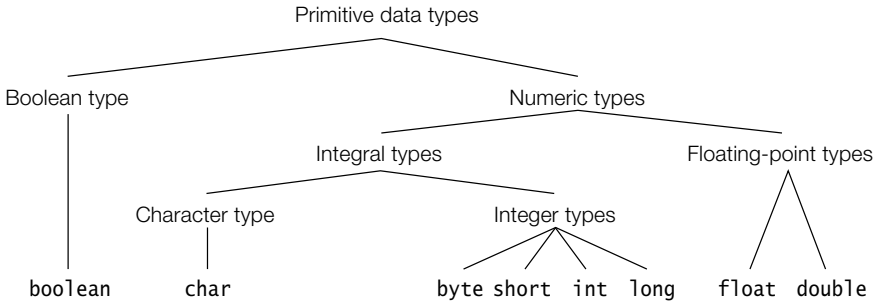
2.2 Primitive Data Types

Figure 2.1 gives an overview of the primitive data types in Java.

Primitive data types in Java can be divided into three main categories:

- *Integral types*—represent signed integers (`byte`, `short`, `int`, `long`) and unsigned character values (`char`)
- *Floating-point types* (`float`, `double`)—represent fractional signed numbers
- *Boolean type* (`boolean`)—represents logical values

Figure 2.1 *Primitive Data Types in Java*



Primitive data values are not objects. Each primitive data type defines the range of values in the data type, and operations on these values are defined by special operators in the language (Chapter 5, p. 143).

Each primitive data type also has a corresponding *wrapper* class that can be used to represent a primitive value as an object. Wrapper classes are discussed in §8.3, p. 346.

The Integer Types

The integer data types are `byte`, `short`, `int`, and `long` (Table 2.10). Their values are signed integers represented by two's complement (§5.5, p. 155).

Table 2.10 *Range of Integer Values*

Data type	Width (bits)	Minimum value MIN_VALUE	Maximum value MAX_VALUE
`byte`	8	-2^7 `(-128)`	2^7-1 `(+127)`
`short`	16	-2^{15} `(-32768)`	$2^{15}-1$ `(+32767)`
`int`	32	-2^{31} `(-2147483648)`	$2^{31}-1$ `(+2147483647)`
`long`	64	-2^{63} `(-9223372036854775808L)`	$2^{63}-1$ `(+9223372036854775807L)`

The `char` Type

The data type `char` represents characters (Table 2.11). Their values are unsigned integers that denote all of the 65536 (2^{16}) characters in the 16-bit Unicode character set. This set includes letters, digits, and special characters.

Table 2.11 *Range of Character Values*

Data type	Width (bits)	Minimum Unicode value	Maximum Unicode value
`char`	16	`0x0 (\u0000)`	`0xffff (\uffff)`

The first 128 characters of the Unicode set are the same as the 128 characters of the 7-bit ASCII character set, and the first 256 characters of the Unicode set correspond to the 256 characters of the 8-bit ISO Latin-1 character set.

The integer types and the `char` type are collectively called *integral types*.

The Floating-Point Types

Floating-point numbers are represented by the `float` and `double` data types.

Floating-point numbers conform to the IEEE 754-1985 binary floating-point standard. Table 2.12 shows the range of values for positive floating-point numbers, but these apply equally to negative floating-point numbers with the minus sign (-) as a prefix. Zero can be either `0.0` or `-0.0`.

Table 2.12 *Range of Floating-Point Values*

Data type	Width (bits)	Minimum positive value MIN_VALUE	Maximum positive value MAX_VALUE
`float`	32	`1.401298464324817E-45f`	`3.402823476638528860e+38f`
`double`	64	`4.94065645841246544e-324`	`1.79769313486231570e+308`

Since the size for representation is a finite number of bits, certain floating-point numbers can be represented only as approximations. For example, the value of the expression `(1.0/3.0)` is represented as an approximation due to the finite number of bits used to represent floating-point numbers.

The `boolean` Type

The data type `boolean` represents the two logical values denoted by the literals `true` and `false` (Table 2.13).

Table 2.13 *Boolean Values*

Data type	Width	True value literal	False value literal
`boolean`	not applicable	`true`	`false`

Boolean values are produced by all *relational* (§5.11, p. 180), *conditional* (§5.14, p. 186), and *boolean logical operators* (§5.13, p. 184), and are primarily used to govern the flow of control during program execution.

Table 2.14 summarizes the pertinent facts about the primitive data types: their width or size, which indicates the number of bits required to store a primitive value; their range of legal values, which is specified by the minimum and the maximum values permissible; and the name of the corresponding wrapper class (§8.3, p. 346).

Table 2.14 *Summary of Primitive Data Types*

Data type	Width (bits)	Minimum value, maximum value	Wrapper class
`boolean`	not applicable	`true, false`	`Boolean`
`byte`	8	-2^7, 2^7-1	`Byte`
`short`	16	-2^{15}, $2^{15}-1$	`Short`
`char`	16	`0x0, 0xffff`	`Character`
`int`	32	-2^{31}, $2^{31}-1$	`Integer`
`long`	64	-2^{63}, $2^{63}-1$	`Long`

Continues

Table 2.14 *Summary of Primitive Data Types (Continued)*

Data type	Width (bits)	Minimum value, maximum value	Wrapper class
`float`	32	±1.40129846432481707e-45f, ±3.402823476638528860e+38f	`Float`
`double`	64	±4.94065645841246544e-324, ±1.79769313486231570e+308	`Double`

Review Questions

2.5 Which of the following do not denote a primitive data value in Java?

Select the two correct answers.

(a) `"t"`
(b) `'k'`
(c) `50.5F`
(d) `"hello"`
(e) `false`

2.6 Which of the following primitive data types are not integer types?

Select the three correct answers.

(a) `boolean`
(b) `byte`
(c) `float`
(d) `short`
(e) `double`

2.7 Which integral type in Java has the exact range from `-2147483648` (i.e., -2^{31}) to `2147483647` (i.e., $2^{31}-1$), inclusive?

Select the one correct answer.

(a) `byte`
(b) `short`
(c) `int`
(d) `long`
(e) `char`

2.3 Variable Declarations

A *variable* stores a value of a particular type. A variable has a name, a type, and a value associated with it. In Java, variables can store only values of primitive data types and reference values of objects. Variables that store reference values of objects are called *reference variables* (or *object references* or simply *references*).

Declaring and Initializing Variables

Variable declarations are used to specify the type and the name of variables. This implicitly determines their memory allocation and the values that can be stored in them. Examples of declaring variables that can store primitive values follow:

```
char a, b, c;           // a, b and c are character variables.
double area;            // area is a floating-point variable.
boolean flag;           // flag is a boolean variable.
```

The first declaration is equivalent to the following three declarations:

```
char a;
char b;
char c;
```

A declaration can also be combined with an initialization expression to specify an appropriate initial value for the variable. Such declarations are called *declaration statements*.

```
int i = 10,             // i is an int variable with initial value 10.
    j = 0b101;          // j is an int variable with initial value 5.
long big = 2147483648L;  // big is a long variable with specified initial value.
```

Reference Variables

A *reference variable* can store the reference value of an object, and can be used to manipulate the object denoted by the reference value.

A variable declaration that specifies a *reference type* (i.e., a class, an array, an interface name, or an enum type) declares a reference variable. Analogous to the declaration of variables of primitive data types, the simplest form of reference variable declaration specifies the name and the reference type only. The declaration determines which objects can be referenced by a reference variable. Before we can use a reference variable to manipulate an object, it must be declared and initialized with the reference value of the object.

```
Pizza yummyPizza;   // Variable yummyPizza can reference objects of class Pizza.
Hamburger bigOne,   // Variable bigOne can reference objects of class Hamburger,
          smallOne; // and so can variable smallOne.
```

It is important to note that the preceding declarations do not create any objects of class `Pizza` or `Hamburger`. Rather, they simply create variables that can store reference values of objects of the specified classes.

A declaration can also be combined with an initializer expression to create an object whose reference value can be assigned to the reference variable:

```
Pizza yummyPizza = new Pizza("Hot&Spicy"); // Declaration statement
```

The reference variable `yummyPizza` can reference objects of class `Pizza`. The keyword `new`, together with the *constructor call* `Pizza("Hot&Spicy")`, creates an object of the class `Pizza`. The reference value of this object is assigned to the variable `yummyPizza`. The newly created object of class `Pizza` can now be manipulated through the reference variable `yummyPizza`.

2.4 Initial Values for Variables

This section discusses what value, if any, is assigned to a variable when no explicit initial value is provided in the declaration.

Default Values for Fields

Default values for fields of primitive data types and reference types are listed in Table 2.15. The value assigned depends on the type of the field.

Table 2.15 *Default Values*

Data type	Default value
`boolean`	`false`
`char`	`'\u0000'`
Integer (`byte`, `short`, `int`, `long`)	`0L` for `long`, `0` for others
Floating-point (`float`, `double`)	`0.0F` or `0.0D`
Reference types	`null`

If no explicit initialization is provided for a static variable, it is initialized with the default value of its type when the class is loaded. Similarly, if no initialization is provided for an instance variable, it is initialized with the default value of its type when the class is instantiated. The fields of reference types are always initialized with the `null` reference value if no initialization is provided.

Example 2.1 illustrates the default initialization of fields. Note that static variables are initialized when the class is loaded the first time, and instance variables are initialized accordingly in *every* object created from the class `Light`.

Example 2.1 *Default Values for Fields*

```
public class Light {
  // Static variable
  static int counter;      // Default value 0 when class is loaded

  // Instance variables:
  int     noOfWatts = 100; // Explicitly set to 100
  boolean indicator;       // Implicitly set to default value false
  String  location;        // Implicitly set to default value null

  public static void main(String[] args) {
    Light bulb = new Light();
    System.out.println("Static variable counter:     " + Light.counter);
    System.out.println("Instance variable noOfWatts: " + bulb.noOfWatts);
    System.out.println("Instance variable indicator: " + bulb.indicator);
    System.out.println("Instance variable location:  " + bulb.location);
  }
}
```

Output from the program:

```
Static variable counter:     0
Instance variable noOfWatts: 100
Instance variable indicator: false
Instance variable location:  null
```

Initializing Local Variables of Primitive Data Types

Local variables are variables that are declared in methods, constructors, and blocks (Chapter 3, p. 47). They are *not* initialized implicitly when they are allocated memory at method invocation—that is, when the execution of a method begins. The same applies to local variables in constructors and blocks. Local variables must be explicitly initialized before being used. The compiler will report an error only if an attempt is made to *use* an uninitialized local variable.

Example 2.2 *Flagging Uninitialized Local Variables of Primitive Data Types*

```
public class TooSmartClass {
  public static void main(String[] args) {
    int weight = 10, thePrice;                          // (1) Local variables

    if (weight <  10) thePrice = 1000;
    if (weight >  50) thePrice = 5000;
    if (weight >= 10) thePrice = weight*10;             // (2) Always executed
    System.out.println("The price is: " + thePrice);    // (3) Compile-time error!
  }
}
```

In Example 2.2, the compiler complains that the local variable `thePrice` used in the `println` statement at (3) may not be initialized. However, at runtime, the local variable `thePrice` will get the value `100` in the last `if` statement at (2), before it is used in the `println` statement. The compiler does not perform a rigorous analysis of the program in this regard. It compiles the body of a conditional statement only if it can deduce that the condition is `true`. The program will compile correctly if the variable is initialized in the declaration, or if an unconditional assignment is made to the variable.

Replacing the declaration of the local variables at (1) in Example 2.2 with the following declaration solves the problem:

```
int weight = 10, thePrice = 0;          // (1') Both local variables initialized
```

Initializing Local Reference Variables

Local reference variables are bound by the same initialization rules as local variables of primitive data types.

Example 2.3 *Flagging Uninitialized Local Reference Variables*

```
public class VerySmartClass {
  public static void main(String[] args) {
    String importantMessage;       // Local reference variable

    System.out.println("The message length is: " +
                       importantMessage.length());  // Compile-time error!
  }
}
```

In Example 2.3, the compiler complains that the local variable `importantMessage` used in the `println` statement may not be initialized. If the variable `importantMessage` is set to the value `null`, the program will compile. However, a runtime error (`NullPointerException`) will occur when the code is executed, because the variable `importantMessage` will not denote any object. The golden rule is to ensure that a reference variable, whether local or not, is assigned a reference value denoting an object before it is used—that is, to ensure that it does not have the value `null`.

The program compiles and runs if we replace the declaration with the following declaration of the local variable, which creates a string literal and assigns its reference value to the local reference variable `importantMessage`:

```
String importantMessage = "Initialize before use!";
```

Arrays and their default values are discussed in §3.4, p. 58.

Lifetime of Variables

The lifetime of a variable—that is, the time a variable is accessible during execution—is determined by the context in which it is declared. The lifetime of a variable, which is also called its *scope*, is discussed in more detail in §4.4, p. 114. We distinguish among the lifetimes of variables in three contexts:

- *Instance variables*—members of a class, which are created for each object of the class. In other words, every object of the class will have its own copies of these variables, which are local to the object. The values of these variables at any given time constitute the *state* of the object. Instance variables exist as long as the object they belong to is in use at runtime.
- *Static variables*—members of a class, but which are not created for any specific object of the class and, therefore, belong only to the class (§4.4, p. 114). They are created when the class is loaded at runtime, and exist as long as the class is available at runtime.
- *Local variables* (also called *method automatic variables*)—declared in methods, constructors, and blocks; and created for each execution of the method, constructor, or block. After the execution of the method, constructor, or block completes, local (non-`final`) variables are no longer accessible.

Review Questions

2.8 Which of the following declarations are valid?

Select the three correct answers.

(a) `char a = '\u0061';`
(b) `char 'a' = 'a';`
(c) `char \u0061 = 'a';`
(d) `ch\u0061r a = 'a';`
(e) `ch'a'r a = 'a';`

2.9 Given the following code within a method, which statement is true?

```
int i, j;
j = 5;
```

Select the one correct answer.

(a) Local variable `i` is not declared.
(b) Local variable `j` is not declared.
(c) Local variable `i` is declared but not initialized.
(d) Local variable `j` is declared but not initialized.
(e) Local variable `j` is initialized but not declared.

2.10 In which of these variable declarations will the variable remain uninitialized unless it is explicitly initialized?

Select the one correct answer.

(a) Declaration of an instance variable of type `int`
(b) Declaration of a static variable of type `float`
(c) Declaration of a local variable of type `float`
(d) Declaration of a static variable of type `Object`
(e) Declaration of an instance variable of type `int[]`

2.11 What will be the result of compiling and running the following program?

```
public class Init {

  String title;
  boolean published;

  static int total;
  static double maxPrice;

  public static void main(String[] args) {
    Init initMe = new Init();
    double price;
    if (true)
      price = 100.00;
```

```
    System.out.println("|" + initMe.title + "|" + initMe.published + "|" +
                       Init.total + "|" + Init.maxPrice + "|" + price + "|");
  }
}
```

Select the one correct answer.

(a) The program will fail to compile.
(b) The program will compile, and print `|null|false|0|0.0|0.0|` at runtime.
(c) The program will compile, and print `|null|true|0|0.0|100.0|` at runtime.
(d) The program will compile, and print `| |false|0|0.0|0.0|` at runtime.
(e) The program will compile, and print `|null|false|0|0.0|100.0|` at runtime.

Chapter Summary

The following topics were covered in this chapter:

- Basic language elements: identifiers, keywords, separators, literals, whitespace, and comments
- Primitive data types: integral, floating-point, and boolean
- Notational representation of numbers in decimal, binary, octal, and hexadecimal systems
- Declaration and initialization of variables, including reference variables
- Usage of default values for instance variables and static variables
- Lifetime of instance variables, static variables, and local variables

Programming Exercise

2.1 The following program has several errors. Modify the program so that it will compile and run without errors.

```
// File: Temperature.java
PUBLIC CLASS temperature {
  PUBLIC void main(string args) {
    double fahrenheit = 62.5;
    */ Convert /*
    double celsius = f2c(fahrenheit);
    System.out.println(fahrenheit + 'F' + " = " + Celsius + 'C');
  }

  double f2c(float fahr) {
    RETURN (fahr - 32.0) * 5.0 / 9.0;
  }

}
```