Introduction to Computer Programming

Java Summary

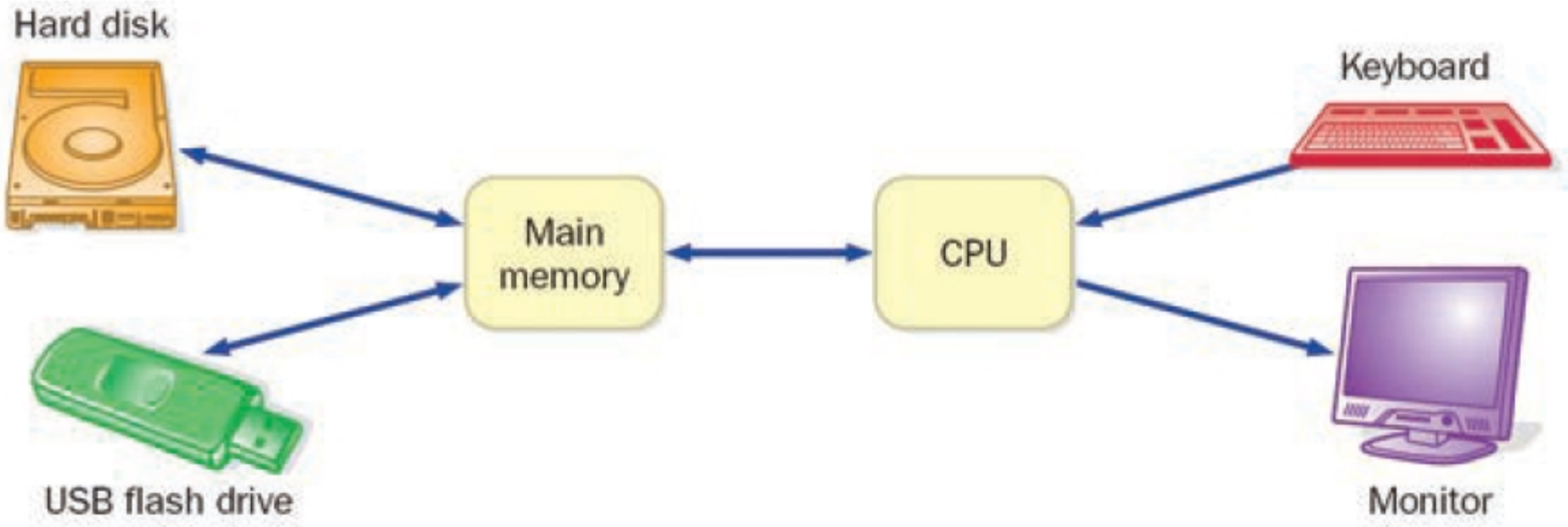


FIGURE 1.1 A simplified view of a computer system

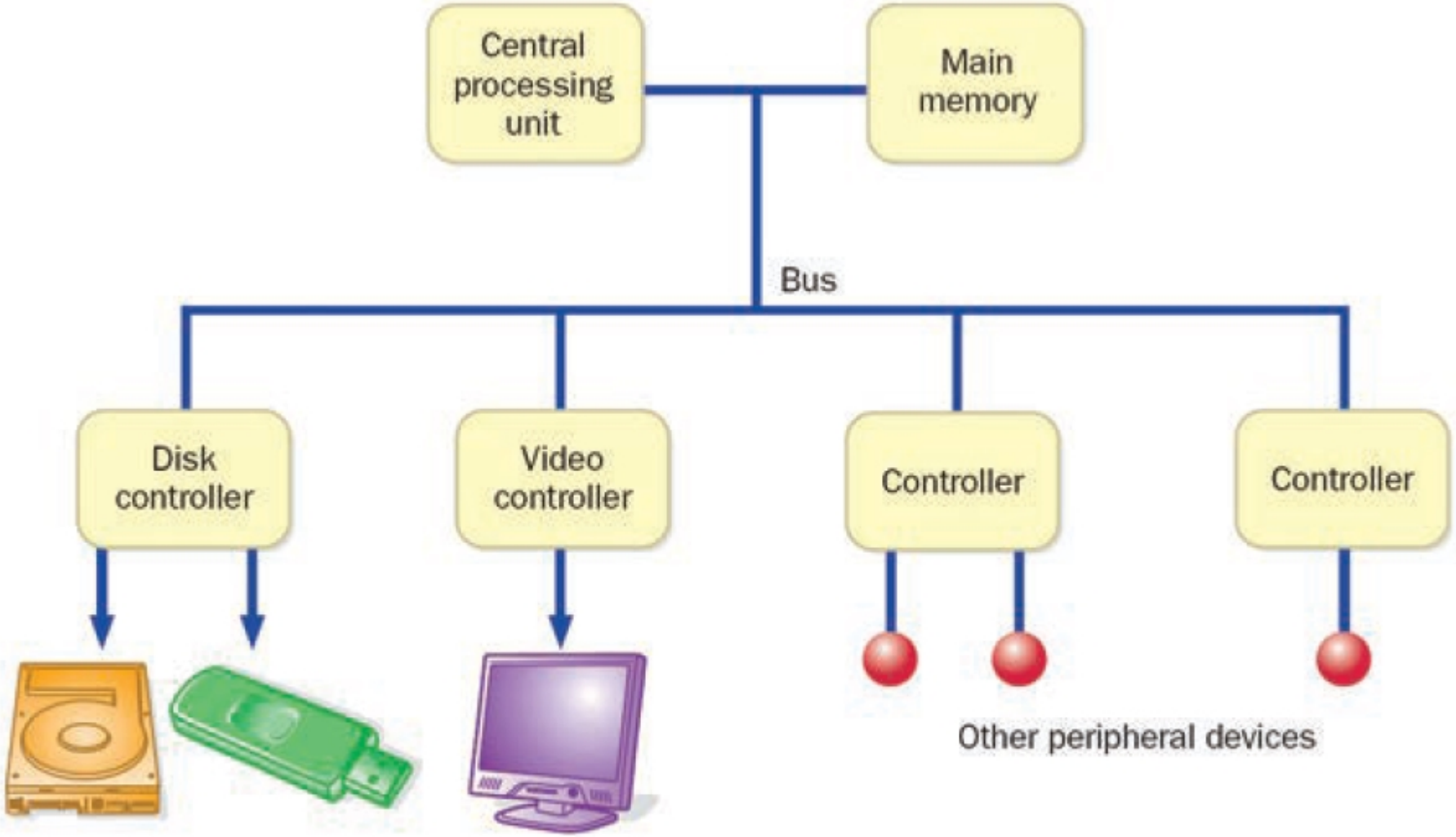
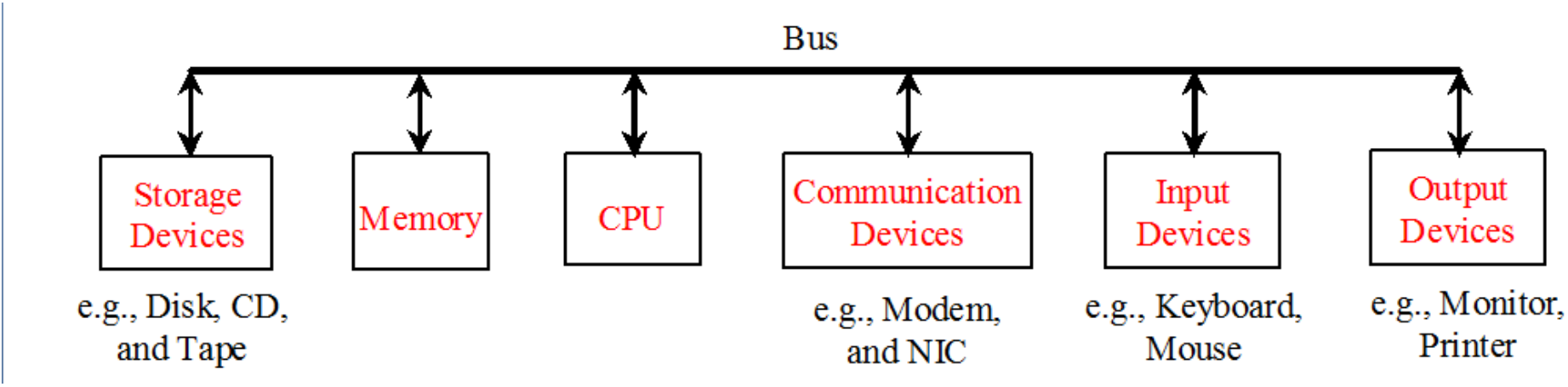


FIGURE 1.9 Basic computer architecture

Hardware

- A computer consists of various devices referred to as hardware
  - e.g., the keyboard, screen, mouse, disks, memory, DVD, CD-ROM and central processing units (CPU)



Software

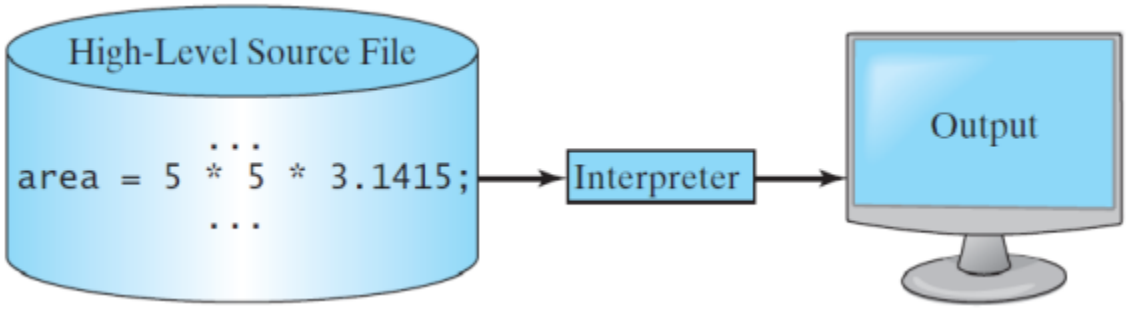
- The programs that run on a computer with data and documents are referred to as software.

What is a program?

- A sequence of instructions that specifies how to perform a computation.
- Instructions or statements perform:
  - input: get data
  - output: output data
  - math: perform math operations
  - testing: check for conditions, run statements
  - repetition: perform actions

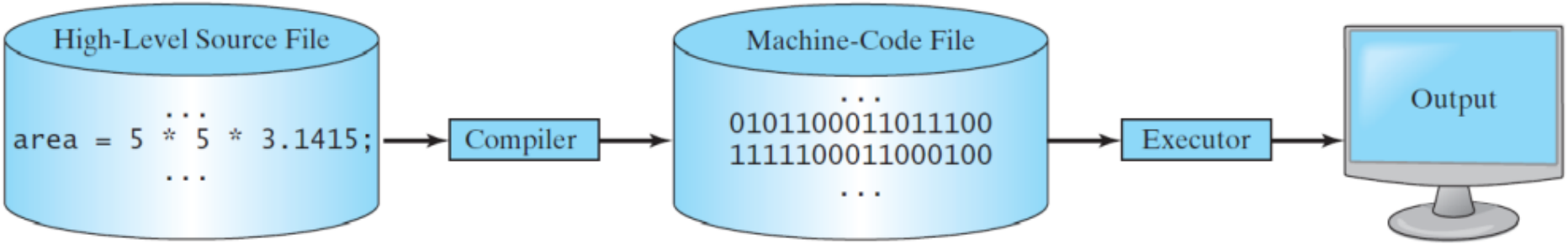
Interpreting Source Code

An interpreter reads one statement from the source code, translates it to the machine code or virtual machine code, and then executes it right away.



Compiling Source Code

A compiler translates the entire source code into a machine-code file, and the machine-code file is then executed.



Java is both compiled and interpreted.

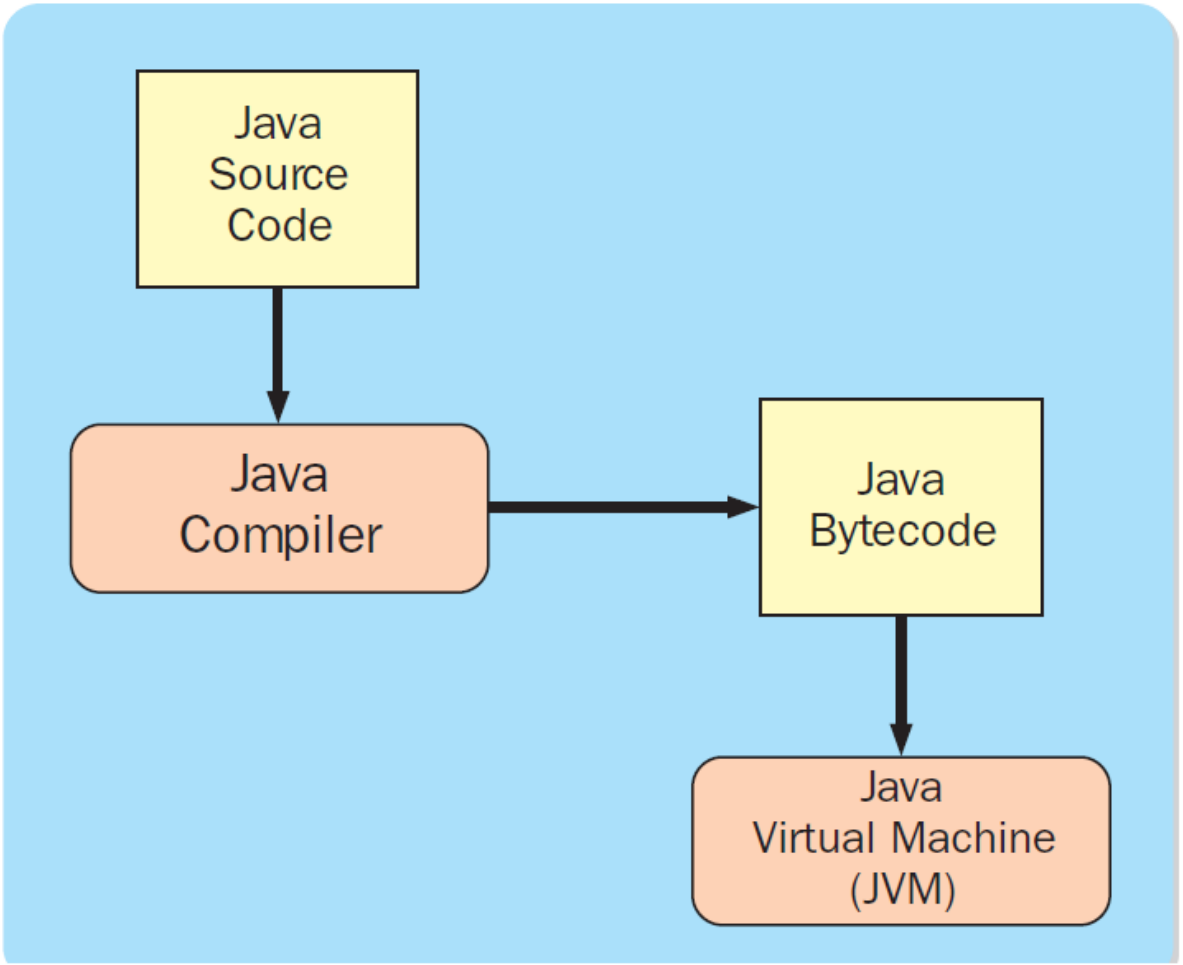


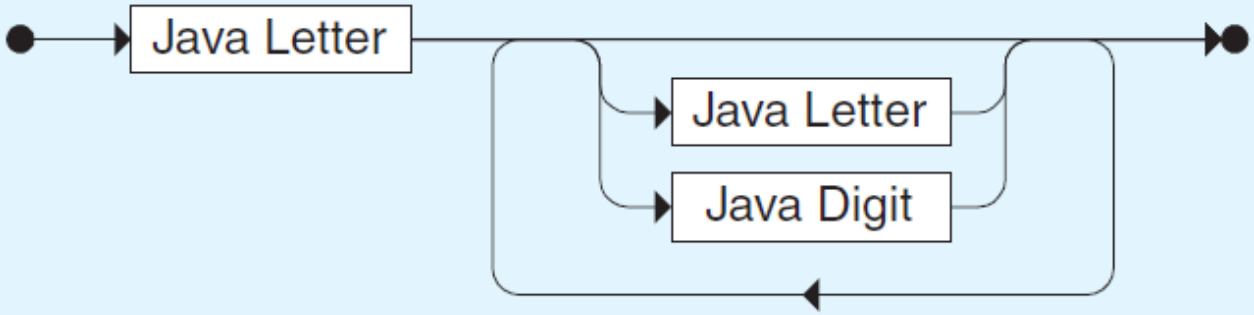
FIGURE 1.21 The Java translation and execution process

Version	Year	New Features
1.0	1996	Initial deployment
1.1	1997	Inner classes
1.2	1998	Collections framework, Swing graphics
1.3	2000	Sound framework
1.4	2002	Assertions, XML support, regular expressions
5	2004	Generics, for-each loop, autoboxing, enumerations, annotations, variable-length parameter lists
6	2006	GUI improvements, various library updates
7	2010	Use of strings in a switch, other language changes
8	2014	(Probable) Lambda expressions, date and time API

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

FIGURE 1.18 Java reserved words

Identifier



An identifier is a letter followed by zero or more letters and digits. A Java Letter includes the 26 English alphabetic characters in both uppercase and lowercase, the $ and _ (underscore) characters, as well as alphabetic characters from other languages. A Java Digit includes the digits 0 through 9.

High-Level Language	Assembly Language	Machine Language
a + b	ld [%fp–20], %o0	...
	ld [%fp–24], %o1	1101 0000 0000 0111
	add %o0, %o1, %o0	1011 1111 1110 1000
		1101 0010 0000 0111
		1011 1111 1110 1000
		1001 0000 0000 0000
		...

FIGURE 1.19 A high-level expression and its assembly language and machine language equivalent

1 bit 2 items	2 bits 4 items	3 bits 8 items	4 bits 16 items	5 bits 32 items	
0	00	000	0000	00000	10000
1	01	001	0001	00001	10001
	10	010	0010	00010	10010
	11	011	0011	00011	10011
		100	0100	00100	10100
		101	0101	00101	10101
		110	0110	00110	10110
		111	0111	00111	10111
			1000	01000	11000
			1001	01001	11001
			1010	01010	11010
			1011	01011	11011
			1100	01100	11100
			1101	01101	11101
			1110	01110	11110
			1111	01111	11111

FIGURE 1.7 The number of bits used determines the number of items that can be represented.

Binary (base 2)	Octal (base 8)	Decimal (base 10)	Hexadecimal (base 16)
0	0	0	0
1	1	1	1
10	2	2	2
11	3	3	3
100	4	4	4
101	5	5	5
110	6	6	6
111	7	7	7
1000	10	8	8
1001	11	9	9
1010	12	10	A
1011	13	11	B
1100	14	12	C
1101	15	13	D
1110	16	14	E
1111	17	15	F
10000	20	16	10
10001	21	17	11
10010	22	18	12
10011	23	19	13
10100	24	20	14

FIGURE B.4 Counting in various number systems

Unit	Symbol	Number of Bytes
byte		$2^0 = 1$
kilobyte	KB	$2^{10} = 1024$
megabyte	MB	$2^{20} = 1{,}048{,}576$
gigabyte	GB	$2^{30} = 1{,}073{,}741{,}824$
terabyte	TB	$2^{40} = 1{,}099{,}511{,}627{,}776$
petabyte	PB	$2^{50} = 1{,}125{,}899{,}906{,}842{,}624$

FIGURE 1.11 Units of binary storage

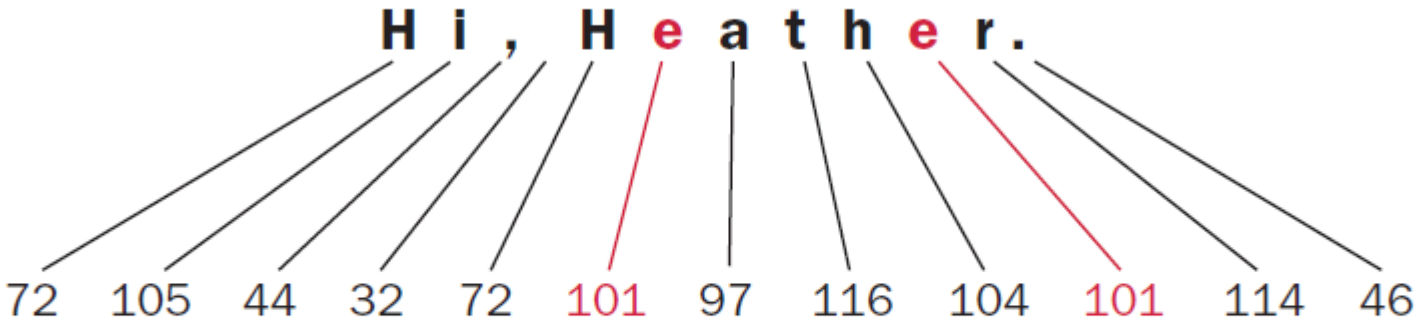


FIGURE 1.5 Text is stored by mapping each character to a number

Value	Char	Value	Char	Value	Char	Value	Char	Value	Char
32	*space*	51	3	70	F	89	Y	108	l
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	–	114	r
39	'	58	:	77	M	96	'	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	'	63	?	82	R	101	e	120	x
45	–	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~

FIGURE C.1 A small portion of the Unicode character set

TABLE 4.4 ASCII Code for Commonly Used Characters

Characters	*Code Value in Decimal*	*Unicode Value*
`'0'` to `'9'`	48 to 57	\u0030 to \u0039
`'A'` to `'Z'`	65 to 90	\u0041 to \u005A
`'a'` to `'z'`	97 to 122	\u0061 to \u007A

TABLE 4.5 Escape Sequences

Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

Value	Character	Source
1071	Я	Russian (Cyrillic)
3593	ค	Thai
5098	Ꮺ	Cherokee
8478	℞	Letterlike Symbols
8652	⇌	Arrows
10287	⠯	Braille
13407	兵	Chinese/Japanese/Korean (Common)

FIGURE C.3 Some non-Western characters in the Unicode character set

TABLE 4.6 Methods in the Character Class

Method	*Description*
`isDigit(ch)`	Returns true if the specified character is a digit.
`isLetter(ch)`	Returns true if the specified character is a letter.
`isLetterOfDigit(ch)`	Returns true if the specified character is a letter or digit.
`isLowerCase(ch)`	Returns true if the specified character is a lowercase letter.
`isUpperCase(ch)`	Returns true if the specified character is an uppercase letter.
`toLowerCase(ch)`	Returns the lowercase of the specified character.
`toUpperCase(ch)`	Returns the uppercase of the specified character.

TABLE 4.7 Simple Methods for `String` Objects

Method	*Description*
`length()`	Returns the number of characters in this string.
`charAt(index)`	Returns the character at the specified index from this string.
`concat(s1)`	Returns a new string that concatenates this string with string `s1`.
`toUpperCase()`	Returns a new string with all letters in uppercase.
`toLowerCase()`	Returns a new string with all letters in lowercase
`trim()`	Returns a new string with whitespace characters trimmed on both sides.

TABLE 4.8 Comparison Methods for `String` Objects

Method	*Description*
`equals(s1)`	Returns true if this string is equal to string `s1`.
`equalsIgnoreCase(s1)`	Returns true if this string is equal to string `s1`; it is case insensitive.
`compareTo(s1)`	Returns an integer greater than 0, equal to 0, or less than 0 to indicate whether this string is greater than, equal to, or less than `s1`.
`compareToIgnoreCase(s1)`	Same as `compareTo` except that the comparison is case insensitive.
`startsWith(prefix)`	Returns true if this string starts with the specified prefix.
`endsWith(suffix)`	Returns true if this string ends with the specified suffix.
`contains(s1)`	Returns true if `s1` is a substring in this string.

TABLE 4.9 The `String` class contains the methods for obtaining substrings.

Method	*Description*
`substring(beginIndex)`	Returns this string's substring that begins with the character at the specified `beginIndex` and extends to the end of the string, as shown in Figure 4.2.
`substring(beginIndex, endIndex)`	Returns this string's substring that begins at the specified `beginIndex` and extends to the character at index `endIndex - 1`, as shown in Figure 4.2. Note that the character at `endIndex` is not part of the substring.

TABLE 4.10 The `String` class contains the methods for finding substrings.

Method	*Description*
`index(ch)`	Returns the index of the first occurrence of `ch` in the string. Returns `-1` if not matched.
`indexOf(ch, fromIndex)`	Returns the index of the first occurrence of `ch` after `fromIndex` in the string. Returns `-1` if not matched.
`indexOf(s)`	Returns the index of the first occurrence of string `s` in this string. Returns `-1` if not matched.
`indexOf(s, fromIndex)`	Returns the index of the first occurrence of string `s` in this string after `fromIndex`. Returns `-1` if not matched.
`lastIndexOf(ch)`	Returns the index of the last occurrence of `ch` in the string. Returns `-1` if not matched.
`lastIndexOf(ch, fromIndex)`	Returns the index of the last occurrence of `ch` before `fromIndex` in this string. Returns `-1` if not matched.
`lastIndexOf(s)`	Returns the index of the last occurrence of string `s`. Returns `-1` if not matched.
`lastIndexOf(s, fromIndex)`	Returns the index of the last occurrence of string `s` before `fromIndex`. Returns `-1` if not matched.

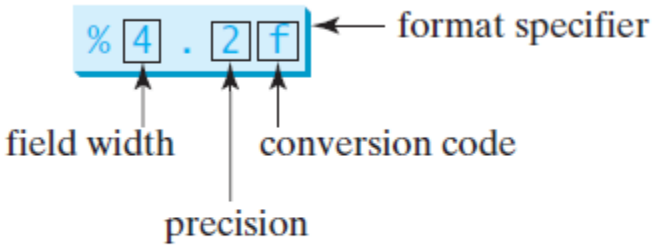
TABLE 4.11 Frequently Used Format Specifiers

Format Specifier	*Output*	*Example*
`%b`	a Boolean value	true or false
`%c`	a character	'a'
`%d`	a decimal integer	200
`%f`	a floating-point number	45.460000
`%e`	a number in standard scientific notation	4.556000e+01
`%s`	a string	"Java is cool"

TABLE 4.12 Examples of Specifying Width and Precision

Example	*Output*
`%5c`	Output the character and add four spaces before the character item, because the width is 5.
`%6b`	Output the Boolean value and add one space before the false value and two spaces before the true value.
`%5d`	Output the integer item with width at least 5. If the number of digits in the item is < 5, add spaces before the number. If the number of digits in the item is > 5, the width is automatically increased.
`%10.2f`	Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus, there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is < 7, add spaces before the number. If the number of digits before the decimal point in the item is > 7, the width is automatically increased.
`%10.2e`	Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.
`%12s`	Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.

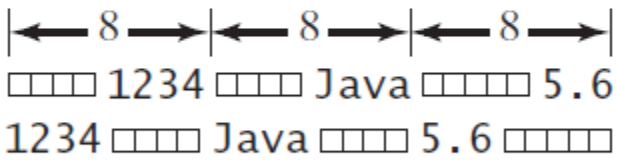
```
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.printf("Interest is $%4.2f",
  interest);
```




```
Interest is $16.40
```

```
System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.63);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.63);
```

display



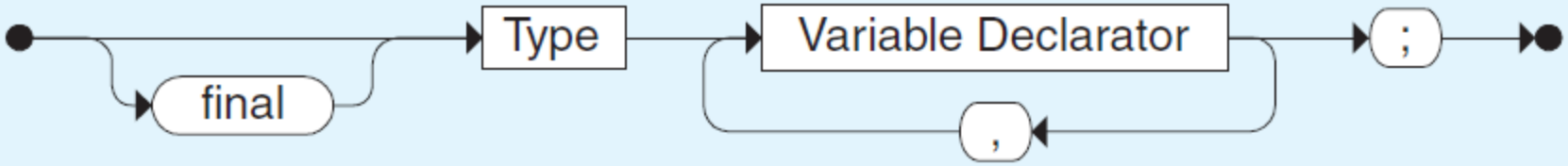
where the square box (□) denotes a blank space.

Object-Oriented Software Principles

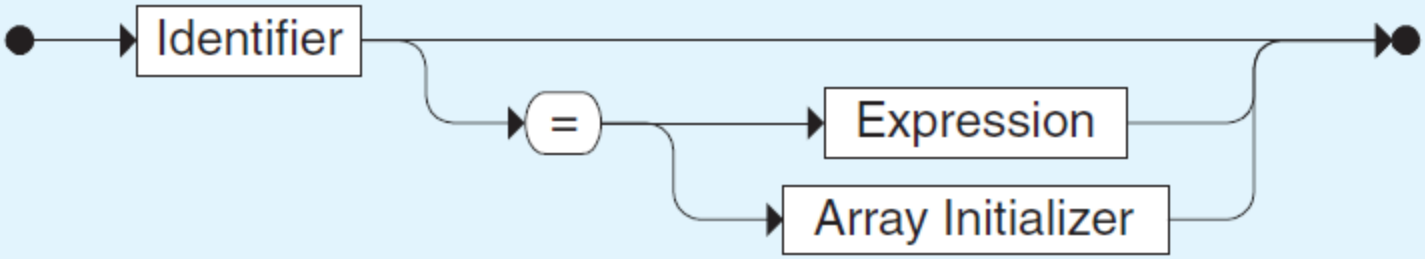
Object-oriented programming ultimately requires a solid understanding of the following terms:

- object
- attribute
- method
- class
- encapsulation
- inheritance
- polymorphism

Local Variable Declaration



Variable Declarator



A variable declaration consists of a Type followed by a list of variables. Each variable can be initialized in the declaration to the value of the specified Expression. If the final modifier precedes the declaration, the identifiers are declared as named constants whose values cannot be changed once set.

Type	Storage	Min Value	Max Value
`byte`	8 bits	–128	127
`short`	16 bits	–32,768	32,767
`int`	32 bits	–2,147,483,648	2,147,483,647
`long`	64 bits	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
`float`	32 bits	Approximately –3.4E+38 with 7 significant digits	Approximately 3.4E+38 with 7 significant digits
`double`	64 bits	Approximately –1.7E+308 with 15 significant digits	Approximately 1.7E+308 with 15 significant digits

FIGURE 2.2 The Java numeric primitive types

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

FIGURE 2.5 Java widening conversions

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Conversion Techniques

In Java, conversions can occur in three ways:

- assignment conversion
- promotion
- casting

Primitive Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
char	Character
boolean	Boolean
void	Void

FIGURE 3.8 Wrapper classes in the Java API

```
Integer(int value)
```
Constructor: creates a new Integer object storing the specified value.

```
byte byteValue()
double doubleValue()
float floatValue()
int intValue()
long longValue()
```
Return the value of this Integer as the corresponding primitive type.

```
static int parseInt(String str)
```
Returns the int corresponding to the value stored in the specified string.

```
static String toBinaryString(int num)
static String tohexString(int num)
static String toOctalString(int num)
```
Returns a string representation of the specified integer value in the corresponding base.

FIGURE 3.9 Some methods of the `Integer` class

```
Scanner(InputStream source)
Scanner(File source)
Scanner(String source)
```
Constructors: sets up the new scanner to scan values from the specified source.

```
String next()
```
Returns the next input token as a character string.

```
String nextLine()
```
Returns all input remaining on the current line as a character string.

```
boolean nextBoolean()
byte nextByte()
double nextDouble()
float nextFloat()
int nextInt()
long nextLong()
short nextShort()
```
Returns the next input token as the indicated type. Throws `InputMismatchException` if the next token is inconsistent with the type.

```
boolean hasNext()
```
Returns true if the scanner has another token in its input.

```
Scanner useDelimiter(String pattern)
Scanner useDelimiter(Pattern pattern)
```
Sets the scanner's delimiting pattern.

```
Pattern delimiter()
```
Returns the pattern the scanner is currently using to match delimiters.

```
String findInLine(String pattern)
String findInLine(Pattern pattern)
```
Attempts to find the next occurrence of the specified pattern, ignoring delimiters.

FIGURE 2.7 Some methods of the Scanner class

Package	Provides support to
java.applet	Create programs (applets) that are easily transported across the Web.
java.awt	Draw graphics and create graphical user interfaces; AWT stands for Abstract Windowing Toolkit.
java.beans	Define software components that can be easily combined into applications.
java.io	Perform a wide variety of input and output functions.
java.lang	General support; it is automatically imported into all Java programs.
java.math	Perform calculations with arbitrarily high precision.
java.net	Communicate across a network.
java.rmi	Create programs that can be distributed across multiple computers; RMI stands for Remote Method Invocation.
java.security	Enforce security restrictions.
java.sql	Interact with databases; SQL stands for Structured Query Language.
java.text	Format text for output.
java.util	General utilities.
javax.swing	Create graphical user interfaces with components that extend the AWT capabilities.
javax.xml.parsers	Process XML documents; XML stands for eXtensible Markup Language.

FIGURE 3.2 Some packages in the Java API

```
import java.util.Random;
```

`Random()`
Constructor: creates a new pseudorandom number generator.

`float nextFloat()`
Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

`int nextInt()`
Returns a random number that ranges over all possible `int` values (positive and negative).

`int nextInt(int num)`
Returns a random number in the range 0 to `num-1`.

FIGURE 3.4 Some methods of the `Random` class

```
static int abs(int num)
   Returns the absolute value of num.

static double acos(double num)

static double asin(double num)

static double atan(double num)
   Returns the arc cosine, arc sine, or arc tangent of num.

static double cos(double angle)

static double sin(double angle)

static double tan(double angle)
   Returns the angle cosine, sine, or tangent of angle, which is measured in
   radians.

static double ceil(double num)
   Returns the ceiling of num, which is the smallest whole number greater than or
   equal to num.

static double exp(double power)
   Returns the value e raised to the specified power.

static double floor(double num)
   Returns the floor of num, which is the largest whole number less than or equal
   to num.

static double pow(double num, double power)
   Returns the value num raised to the specified power.

static double random()
   Returns a random number between 0.0 (inclusive) and 1.0 (exclusive).

static double sqrt(double num)
   Returns the square root of num, which must be positive.
```

FIGURE 3.5 Some methods of the Math class

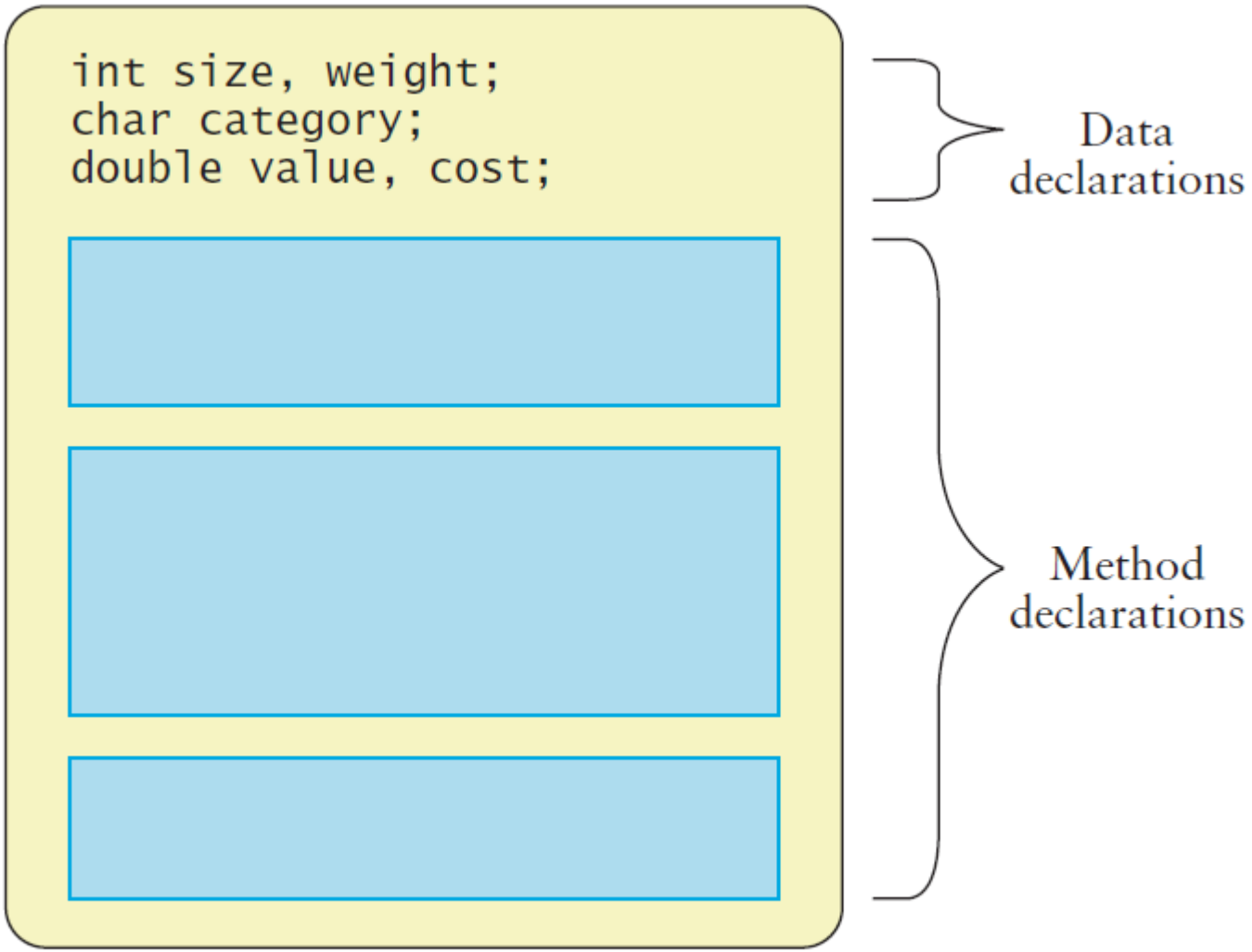


FIGURE 4.2 The members of a class: data and method declarations

Encapsulation

An object should be encapsulated, guarding its data from inappropriate access.

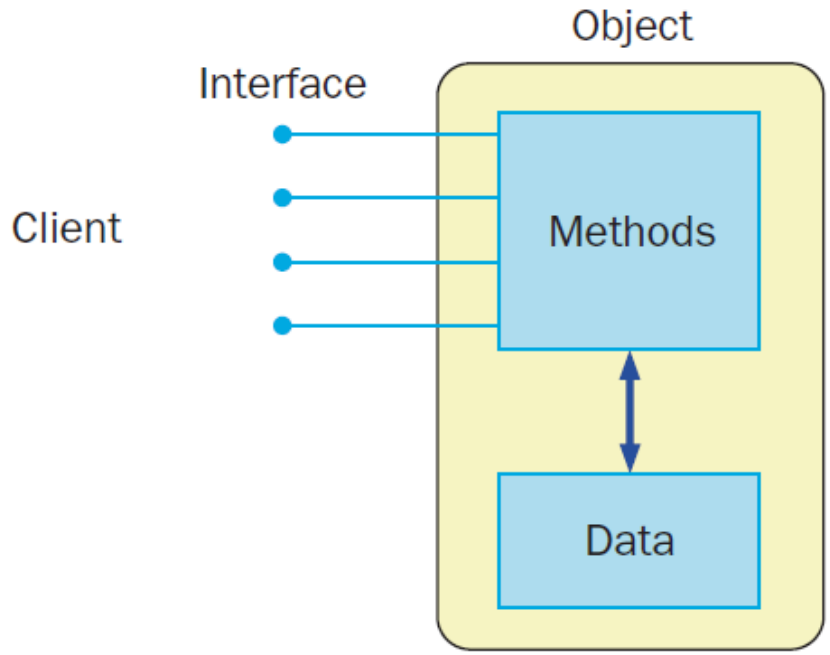


FIGURE 4.5 A client interacting with the methods of an object

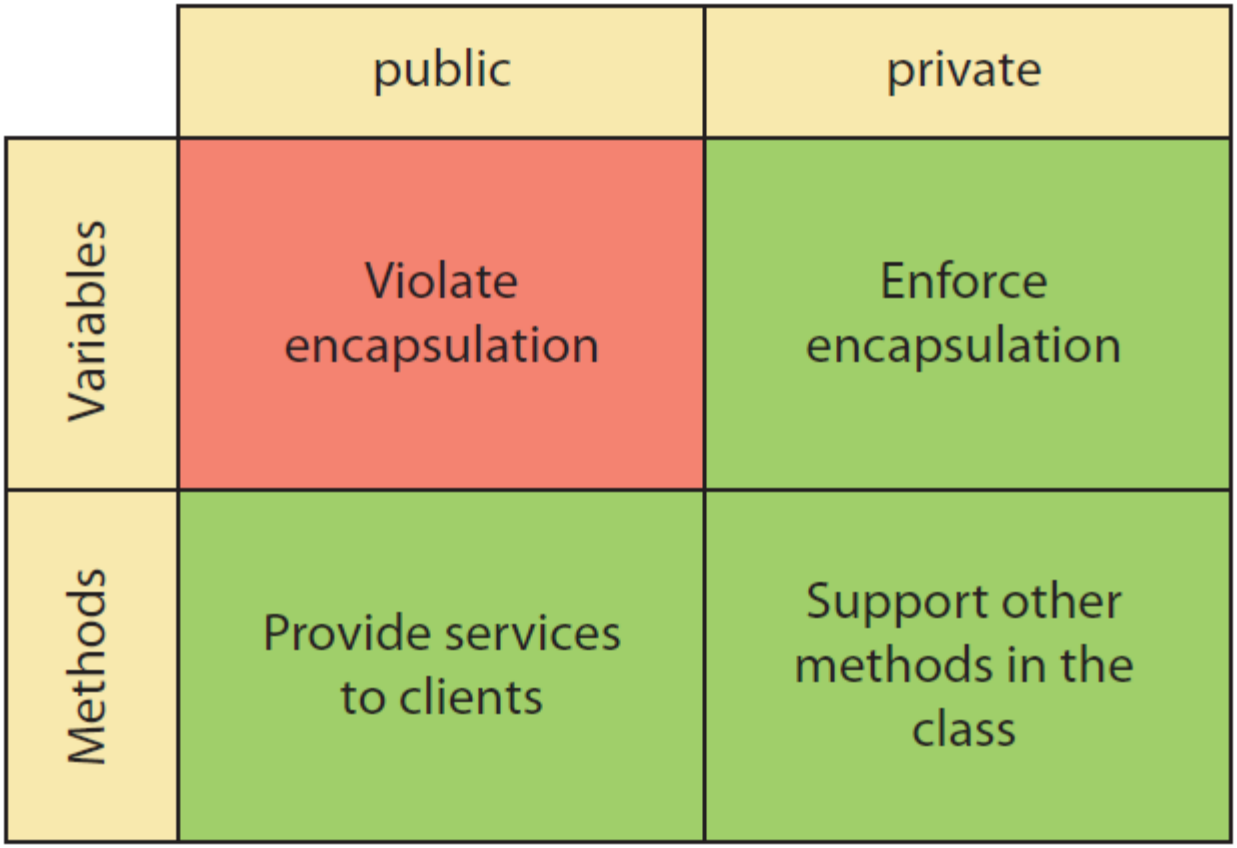
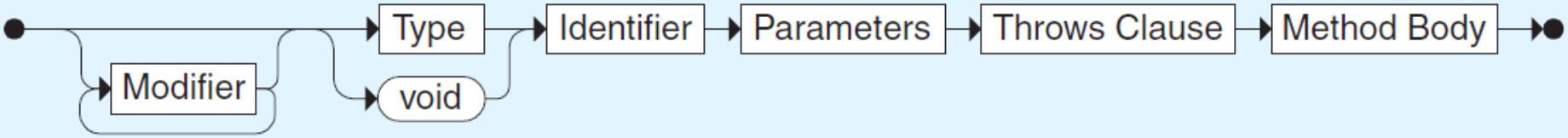
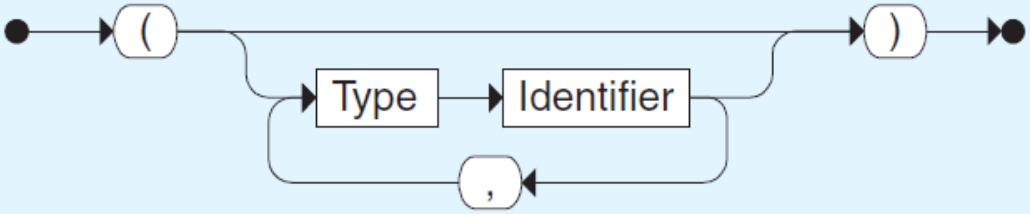


FIGURE 4.6 The effects of public and private visibility

Method Declaration

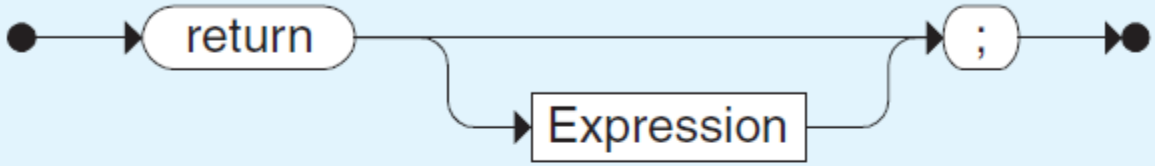


Parameters



A method is defined by optional modifiers, followed by a return Type, followed by an Identifier that determines the method name, followed by a list of Parameters, followed by the Method Body. The return Type indicates the type of value that will be returned by the method, which may be `void`. The Method Body is a block of statements that executes when the method is invoked. The Throws Clause is optional and indicates the exceptions that may be thrown by this method.

Return Statement



A `return` statement consists of the `return` reserved word followed by an optional Expression. When executed, control is immediately returned to the calling method, returning the value defined by Expression.

Examples:

```
return;
return distance * 4;
```

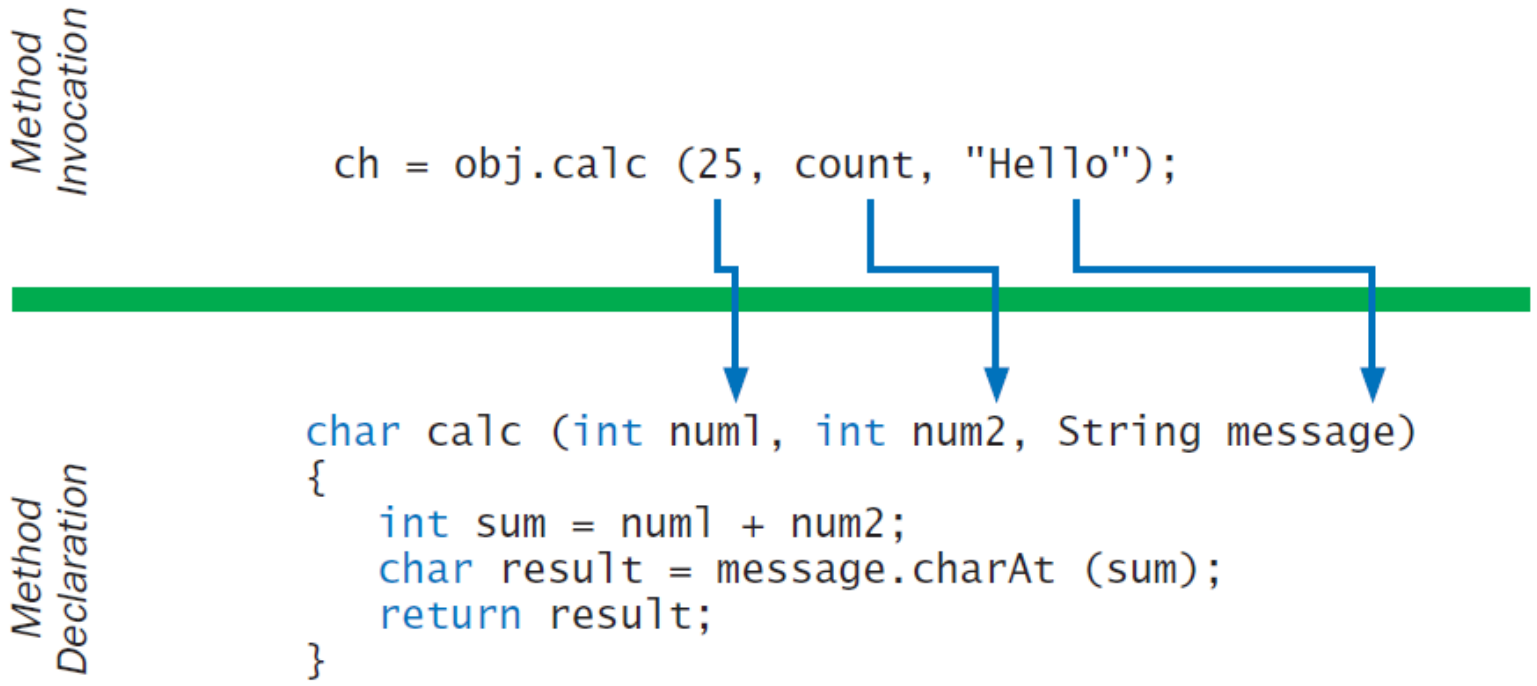


FIGURE 4.8 Passing parameters from the method invocation to the declaration

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

a	!a
false	true
true	false

a	b	a && b	a || b
false	false	false	false
false	true	false	true
true	false	false	true
true	true	true	true

FIGURE 5.1 Java equality and relational operators

Operator	Description	Example	Result
!	logical NOT	! a	true if a is false and false if a is true
&&	logical AND	a && b	true if a and b are both true and false otherwise
||	logical OR	a || b	true if a or b or both are true and false otherwise

FIGURE 5.2 Java logical operators

```
ArrayList<E>()
```
Constructor: creates an initially empty list.

```
boolean add(E obj)
```
Inserts the specified object to the end of this list.

```
void add(int index, E obj)
```
Inserts the specified object into this list at the specified index.

```
void clear()
```
Removes all elements from this list.

```
E remove(int index)
```
Removes the element at the specified index in this list and returns it.

```
E get(int index)
```
Returns the object at the specified index in this list without removing it.

```
int indexOf(Object obj)
```
Returns the index of the first occurrence of the specified object.

```
boolean contains(Object obj)
```
Returns true if this list contains the specified object.

```
boolean isEmpty()
```
Returns true if this list contains no elements.

```
int size()
```
Returns the number of elements in this list.

FIGURE 5.8 Some methods of the `ArrayList<E>` class.

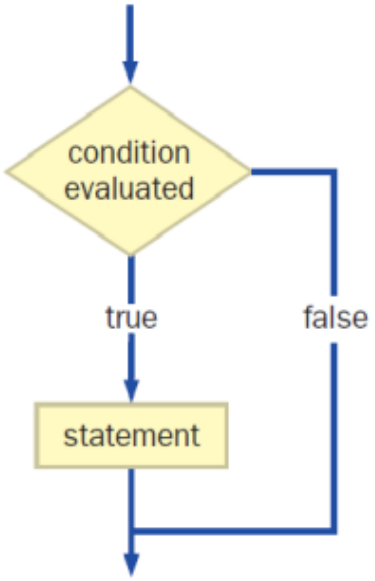
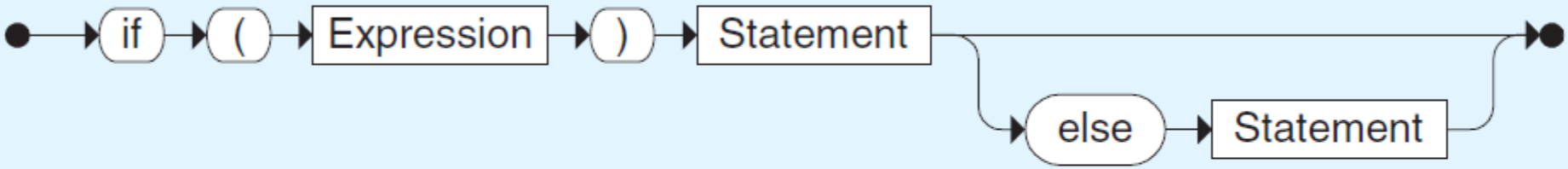


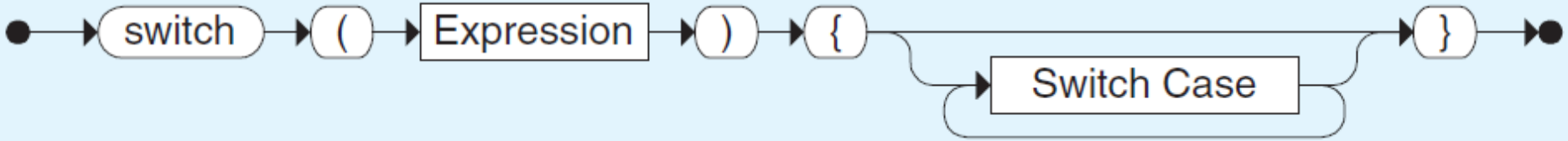
FIGURE 5.6 The logic of an `if` statement

If Statement

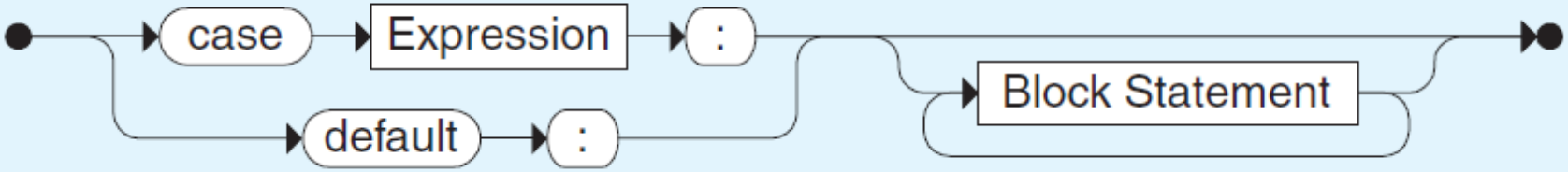


An `if` statement tests the boolean Expression and, if true, executes the first Statement. The optional `else` clause identifies the Statement that should be executed if the Expression is false.

Switch Statement



Switch Case



The `switch` statement evaluates the initial Expression and matches its value with one of the cases. Processing continues with the Statement corresponding to that case. The optional `default` case will be executed if no other case matches.

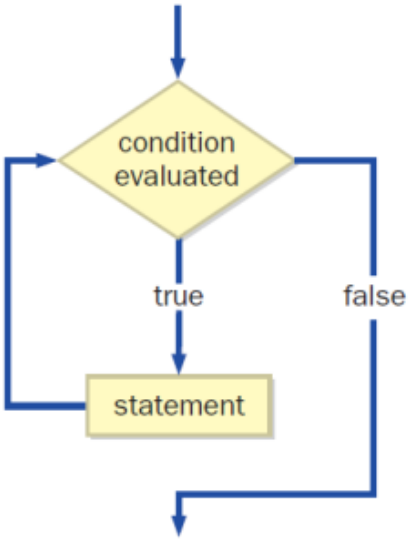


FIGURE 5.7 The logic of a `while` loop

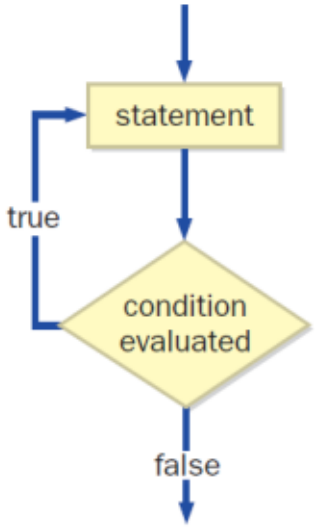


FIGURE 6.1 The logic of a `do` loop

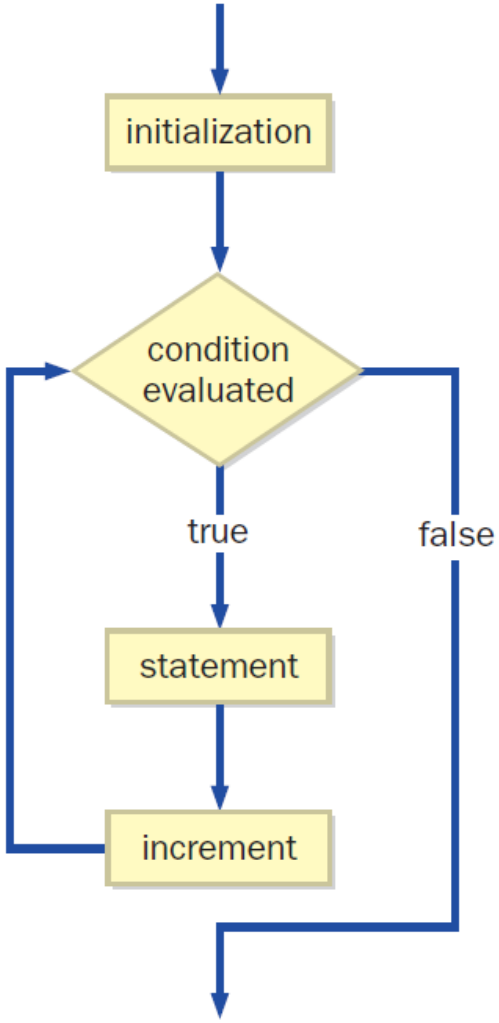
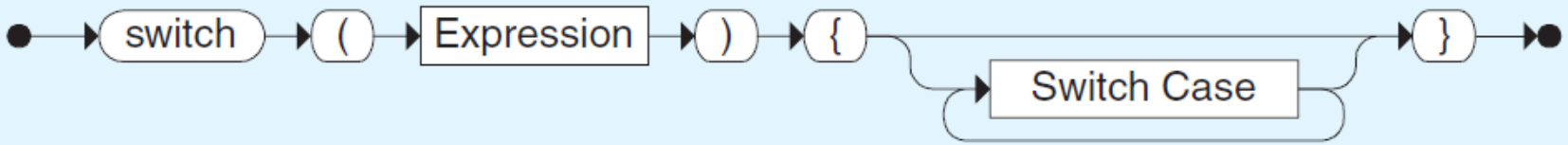


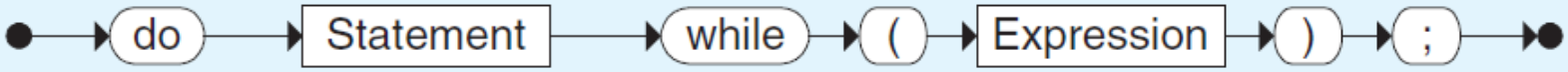
FIGURE 6.2 The logic of a `for` loop

While Statement



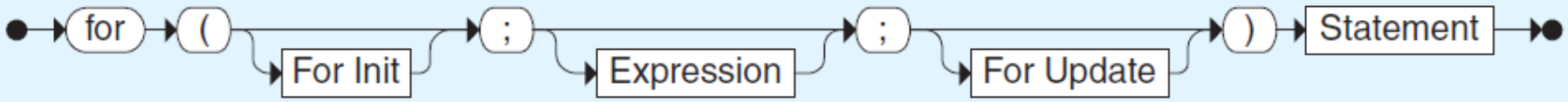
The `while` loop repeatedly executes the specified Statement as long as the boolean Expression is true. The Expression is evaluated first; therefore the Statement might not be executed at all. The Expression is evaluated again after each execution of Statement until the Expression becomes false.

Do Statement

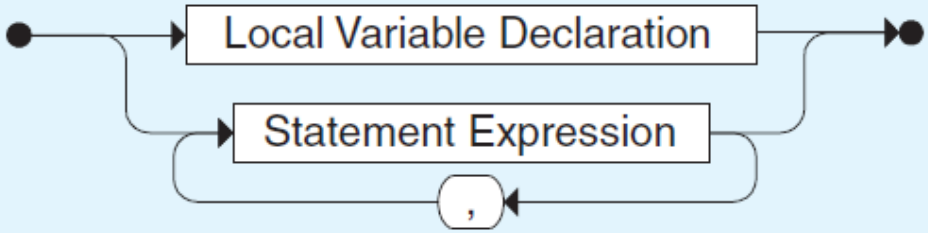


The `do` loop repeatedly executes the specified Statement as long as the boolean Expression is true. The Statement is executed at least once, then the Expression is evaluated to determine whether the Statement should be executed again.

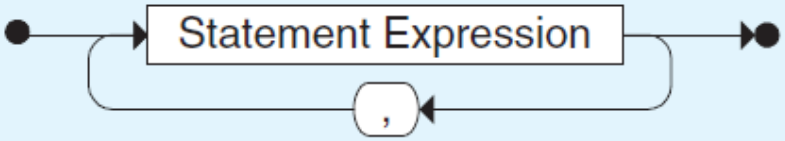
For Statement



For Init



For Update



The `for` statement repeatedly executes the specified Statement as long as the boolean Expression is true. The For Init portion of the header is executed only once, before the loop begins. The For Update portion executes after each execution of Statement.

Examples:

```
for (int value=1; value < 25; value++)
   System.out.println(value + " squared is " + value*value);

for (int num=40; num > 0; num-=3)
   sum = sum + num;
```

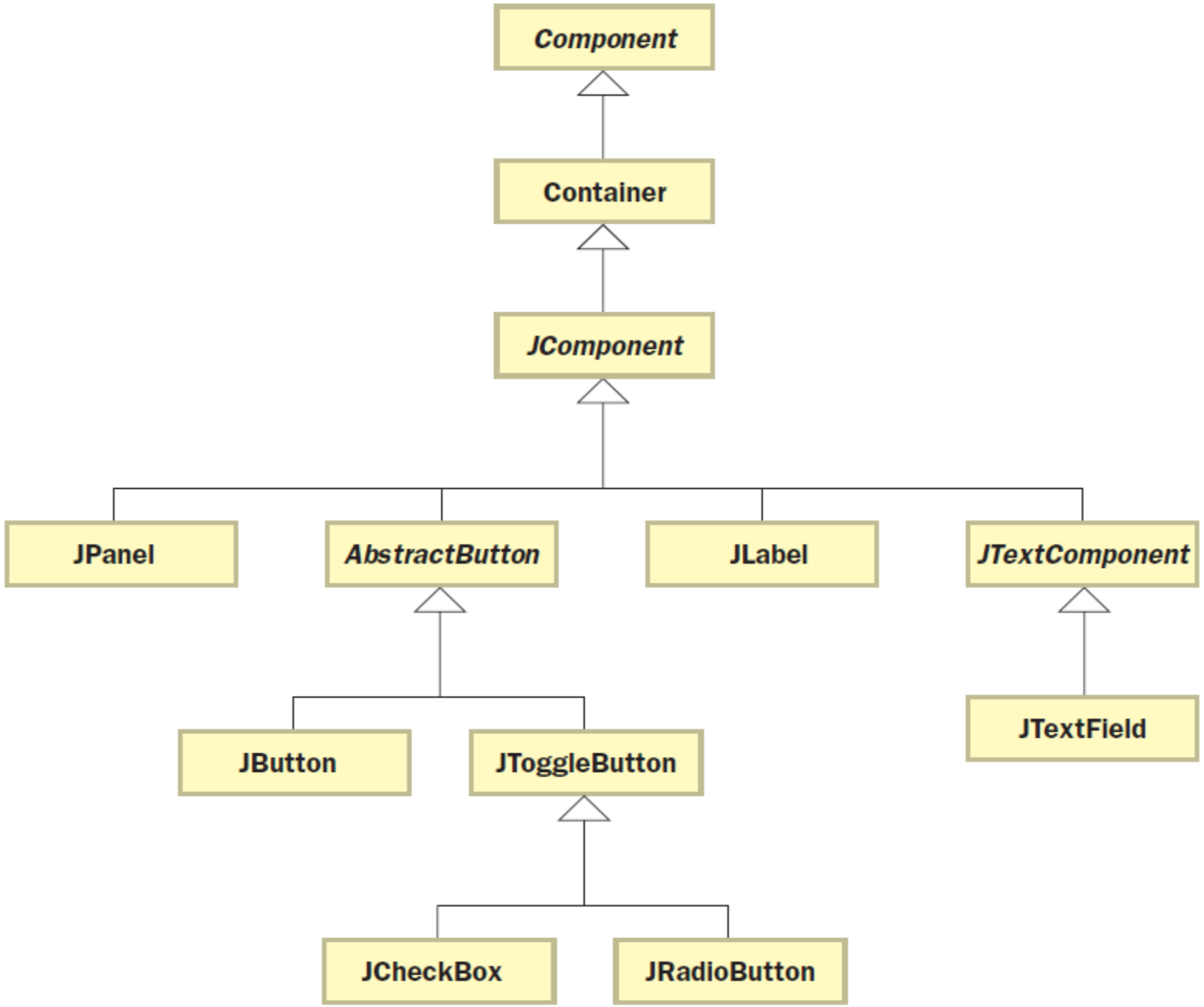


FIGURE 9.7 Part of the GUI component class hierarchy

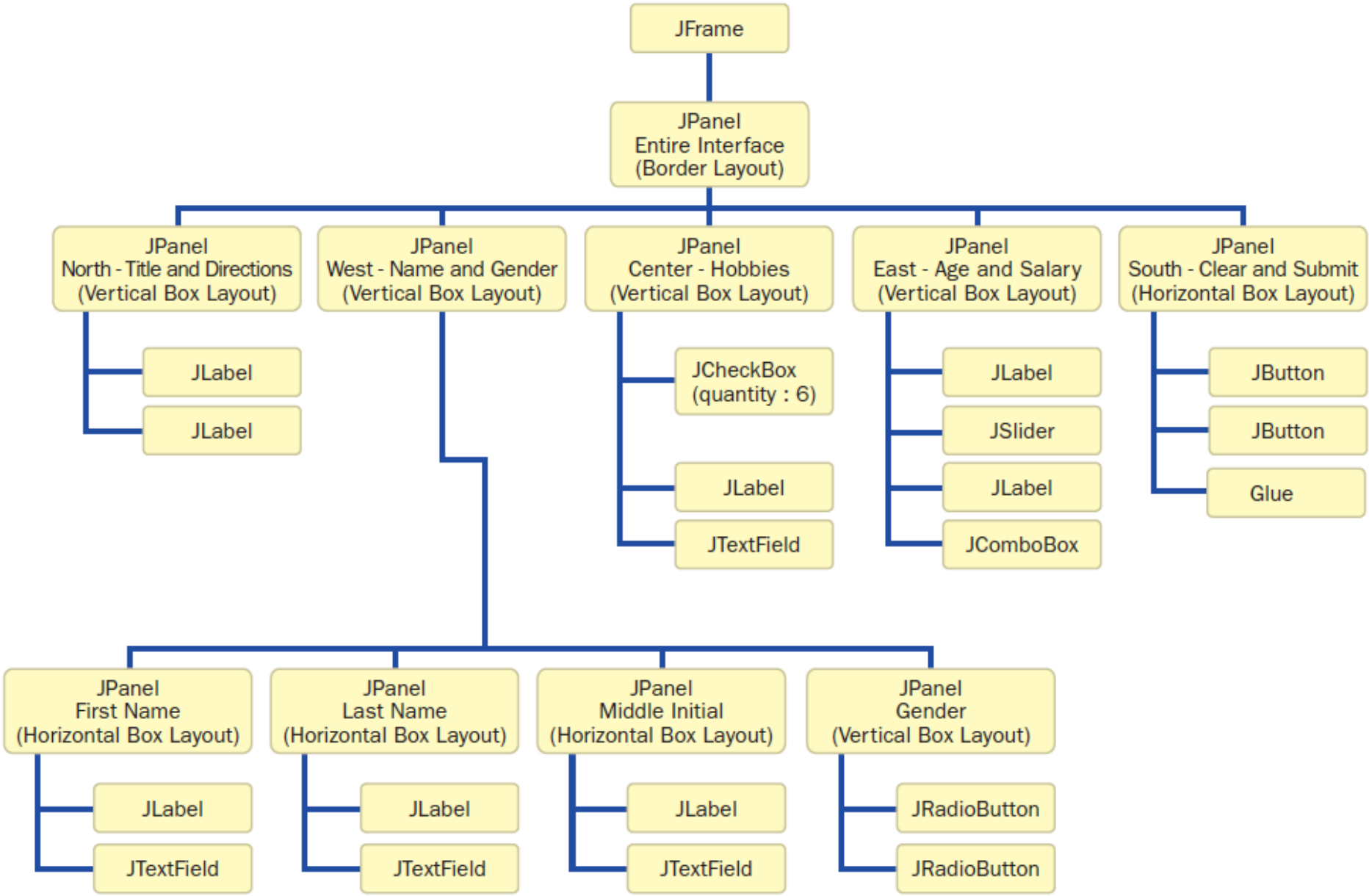


FIGURE 7.13 The containment hierarchy tree

Mouse Event	Description
mouse pressed	The mouse button is pressed down.
mouse released	The mouse button is released.
mouse clicked	The mouse button is pressed down and released without moving the mouse in between.
mouse entered	The mouse pointer is moved onto (over) a component.
mouse exited	The mouse pointer is moved off of a component.

Mouse Motion Event	Description
mouse moved	The mouse is moved.
mouse dragged	The mouse is moved while the mouse button is pressed down.

FIGURE 8.7 Mouse events and mouse motion events

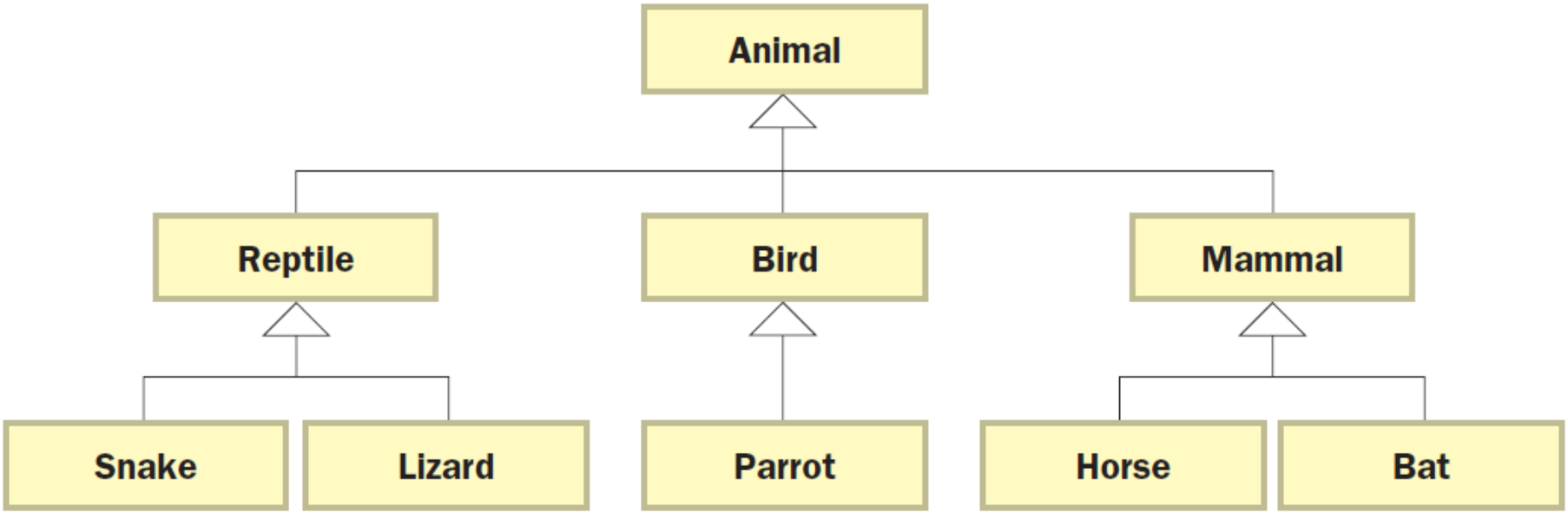


FIGURE 9.3 A UML class diagram showing a class hierarchy

```
boolean equals(Object obj)
```
Returns true if this object is an alias of the specified object.

```
String toString()
```
Returns a string representation of this object.

```
Object clone()
```
Creates and returns a copy of this object.

FIGURE 9.5 Some methods of the `Object` class

```
Timer(int delay, ActionListener listener)
```
Constructor: Creates a timer that generates an action event at regular intervals, specified by the delay. The event will be handled by the specified listener.

```
void addActionListener(ActionListener listener)
```
Adds an action listener to the timer.

```
boolean isRunning()
```
Returns true if the timer is running.

```
void setDelay(int delay)
```
Sets the delay of the timer.

```
void start()
```
Starts the timer, causing it to generate action events.

```
void stop()
```
Stops the timer, causing it to stop generating action events.

FIGURE 9.8 Some methods of the `Timer` class

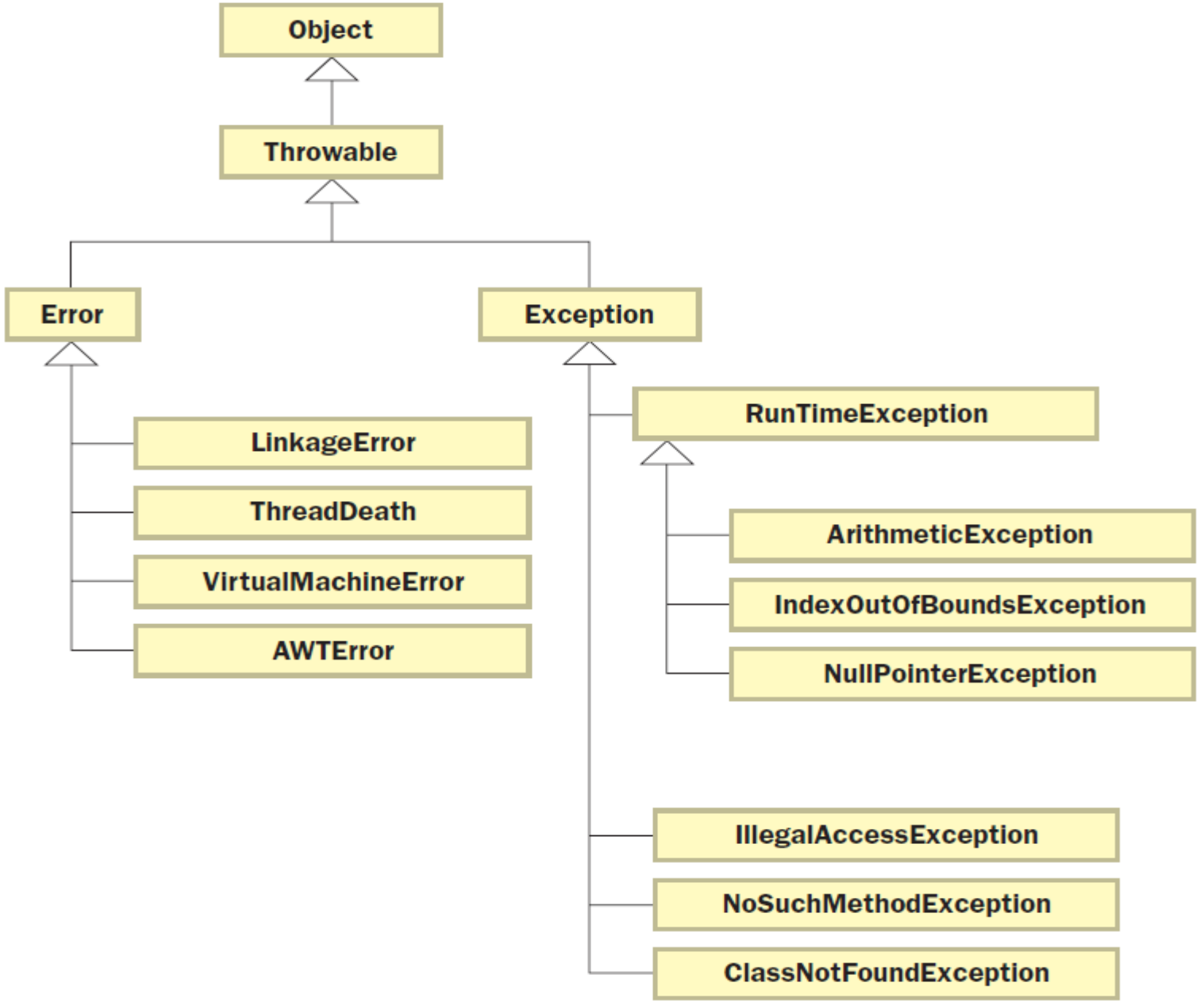


FIGURE 11.1 Part of the Error and Exception class hierarchy

Standard I/O Stream	Description
`System.in`	Standard input stream.
`System.out`	Standard output stream.
`System.err`	Standard error stream (output for error messages)

FIGURE 11.2 Standard I/O streams

Precedence Level	Operator	Operation	Associates
1	[] . (*parameters*) ++ --	array indexing object member reference parameter evaluation and method invocation postfix increment postfix decrement	L to R
2	++ -- + - ~ !	prefix increment prefix decrement unary plus unary minus bitwise NOT logical NOT	R to L
3	new (*type*)	object instantiation cast	R to L
4	* / %	multiplication division remainder	L to R
5	+ + -	addition string concatenation subtraction	L to R
6	<< >> >>>	left shift right shift with sign right shift with zero	L to R

Precedence Level	Operator	Operation	Associates
7	< <= > >= instanceof	less than less than or equal greater than greater than or equal type comparison	L to R
8	== !=	equal not equal	L to R
9	& &	bitwise AND boolean AND	L to R
10	^ ^	bitwise XOR boolean XOR	L to R
11	| |	bitwise OR boolean OR	L to R
12	&&	logical AND	L to R
13	||	logical OR	L to R

Precedence Level	Operator	Operation	Associates
14	?:	conditional operator	R to L
15	=	assignment	R to L
	+=	addition, then assignment	
	+=	string concatenation, then assignment	
	-=	subtraction, then assignment	
	*=	multiplication, then assignment	
	/=	division, then assignment	
	%=	remainder, then assignment	
	<<=	left shift, then assignment	
	>>=	right shift (sign), then assignment	
	>>>=	right shift (zero), then assignment	
	&=	bitwise AND, then assignment	
	&=	boolean AND, then assignment	
	^=	bitwise XOR, then assignment	
	^=	boolean XOR, then assignment	
	|=	bitwise OR, then assignment	
	|=	boolean OR, then assignment	

FIGURE D.1 Java operator precedence, continued

Operator	Description
~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise XOR
<<	left shift
>>	right shift with sign
>>>	right shift with zero fill

FIGURE D.2 Java bitwise operators

num1 & num2	num1 | num2	num1 ^ num2
00101101	00101101	00101101
& 00001110	| 00001110	^ 00001110
= 00001100	= 00101111	= 00100011

FIGURE D.4 Bitwise operations on bytes

a	b	~ a	a & b	a | b	a ^ b
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

FIGURE D.3 Bitwise operations on individual bits

Modifier	Classes and interfaces	Methods and variables
default (no modifier)	Visible in its package.	Visible to any class in the same package as its class.
`public`	Visible anywhere.	Visible anywhere.
`protected`	Can only be applied to inner classes. Visible in its package and to classes that extend the class in which it is declared.	Visible to any class in the same package and to any derived classes.
`private`	Can only be applied to inner classes. Visible to the enclosing class only.	Not visible by any other class.

FIGURE E.1 Java visibility modifiers

Modifier	Class	Interface	Method	Variable
`abstract`	The class may contain abstract methods. It cannot be instantiated.	All interfaces are inherently abstract. The modifier is optional.	No method body is defined. The method requires implementation when inherited.	N/A
`final`	The class cannot be used to drive new classes.	N/A	The method cannot be overridden.	The variable is a constant, whose value cannot be changed once initially set.
`native`	N/A	N/A	No method body is necessary since implementation is in another language.	N/A
`static`	N/A	N/A	Defines a class method. It does not require an instantiated object to be invoked. It cannot reference non-static methods or variables. It is implicitly final.	Defines a class variable. It does not require an instantiated object to be referenced. It is shared (common memory space) among all instances of the class.
`synchronized`	N/A	N/A	The execution of the method is mutually exclusive among all threads.	N/A
`transient`	N/A	N/A	N/A	The variable will not be serialized.
`volatile`	N/A	N/A	N/A	The variable is changed asynchronously. The compiler should not perform optimizations on it.

FIGURE E.3 The rest of the Java modifiers

Important Names

1) Instance **variables**, which is also called **fields**.（实例变量，或叫域）
2) Reference type（引用类型，对应于primitive type）
3) All parameters in Java are **passed by value**

Class Modifier

- The **abstract** class modifier describes a class that has abstract methods. Abstract methods are declared with the **abstract** keyword and are empty (that is, they have no block defining a body of code for this method). A class that has nothing but abstract methods and no instance variables is more properly called an interface, so an **abstract** class usually has a mixture of abstract methods and actual methods. (* abstract class must have abstract methods)
- The **final** class modifier describes a class that can have no subclasses.
- The **public** class modifier describes a class that can be instantiated or extended by anything in the same package or by anything that imports the class. Public classes are declared in their own separate file called classname .java, where “classname” is the name of the class.
- If the **public** class modifier is not used, the class is considered **friendly**. This means that it can be used and instantiated by all classes in the same **package**. This is the

Variable Modifiers

- **public**: Anyone can access public instance variables.
- **protected**: Only methods of the same package or of its subclasses can access protected instance variables.
- **private**: Only methods of the same class (not methods of a subclass) can access private instance variables.
- If none of the modifiers above are used, then the instance variable is **friendly**. Friendly variable can only be access**ed** by methods of classes in the same package.

Additional:

- **static**: The static keyword is used to declare a variable that is associated with the class, not with individual instances of that class. Static variables are used to store "global" information about a class (for example, a static variable could be used to maintain the total number of Gnome objects created). Static variables exist even if no instance of their class is created.
- **final**: A final instance variable is one that must be assigned an initial value, and then• can never be assigned a new value after that. If it is a base type, then it is a constant (like the MAX_HEIGHT constant in the Gnome class). If an object variable is

Method Modifier

•**public**: Anyone can call public methods.

•**protected**: Only methods of the same package or of subclasses can call a protected method.

•**private**: Only methods of the same class (not methods of a subclass) can call a private method.

•If none of the modifiers above are used, then the method is **friendly**. Friendly methods can only be called by objects of classes in the same package.

Additional:

•**abstract**: Amethod declared as abstract has no code. The signature of such a method is followed by a semicolon with no method body. For example:

public abstract void setHeight (double newHeight);

Abstract methods may only appear within an abstract class.

•**final**: This is a method that cannot be overridden by a subclass.

•**static**: This is a method that is associated with the class itself, and not with a particular instance of the class. Static methods can also be used to change the state of static variables associated with a class (provided these variables are not declared to be final).

Limitation of static methods:

- Can only call other static methods.
- Can only access static variable
- Can not use “this” or “super” (“this” is associated with object and

Constructor Modifier

Constructor modifiers follow the same rules as normal methods, except that an **abstract**, **static**, or **final** constructor is not allowed.