Java Fundamentals

Data types
and
Operators

Lexical Issues

Java programs are a collection of whitespace, identifiers, comments, literals, operators, separators, and keywords.

whitespace
identifiers
comments
literals
operators
separators
keyworlds

identifiers

An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers or the underscore and dollar-sign characters.

Some examples of valid identifiers are:

AvgTemp   count   a4 $test   this_ is _ ok

Invalid variable names include:

2count   high-temp   Not/ok

Literals

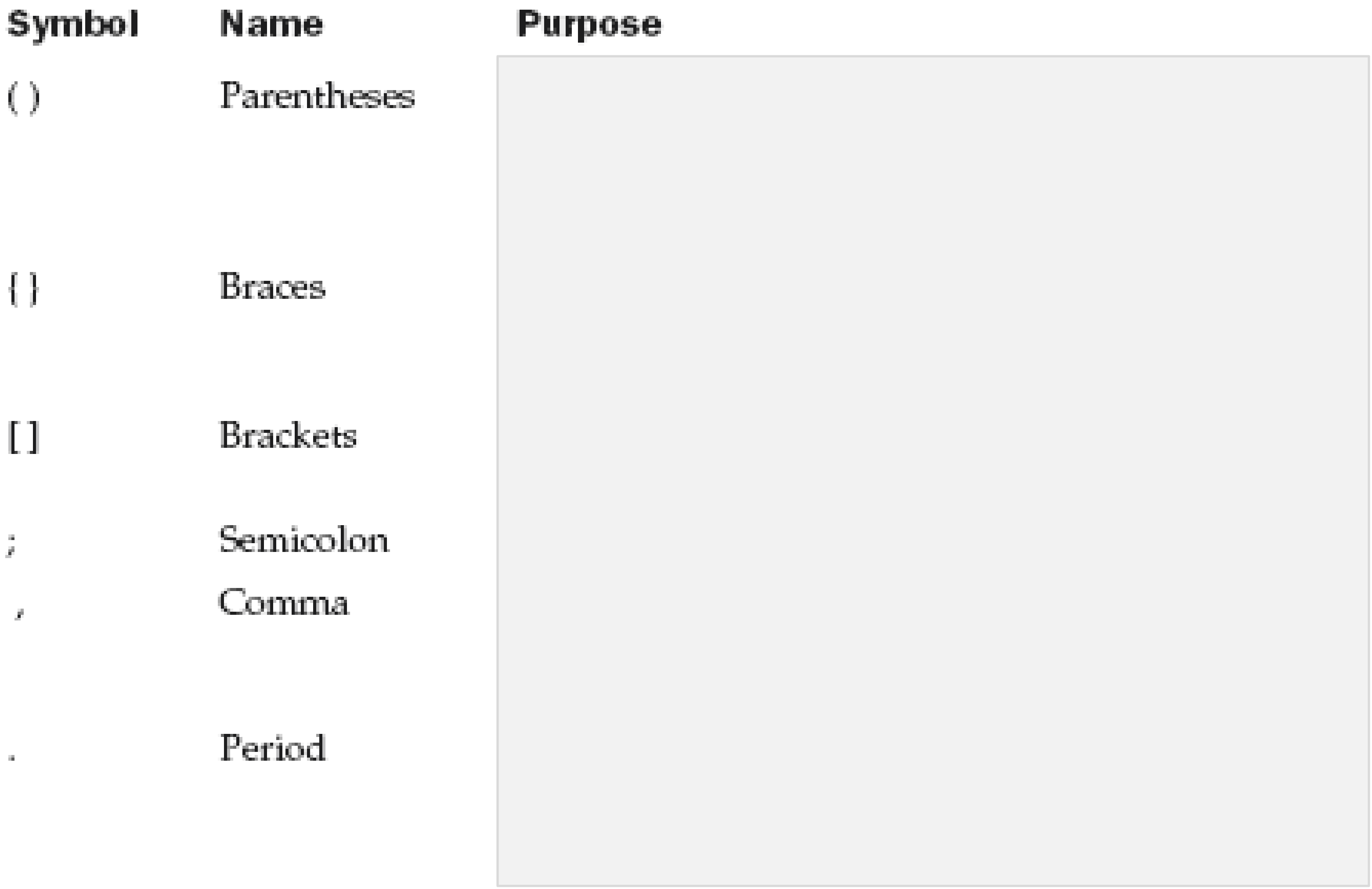
A constant value in Java is created by using a literal representation of it.

For example, here are some literals:
100 98.6 ‘X’ “This is a test”

comments

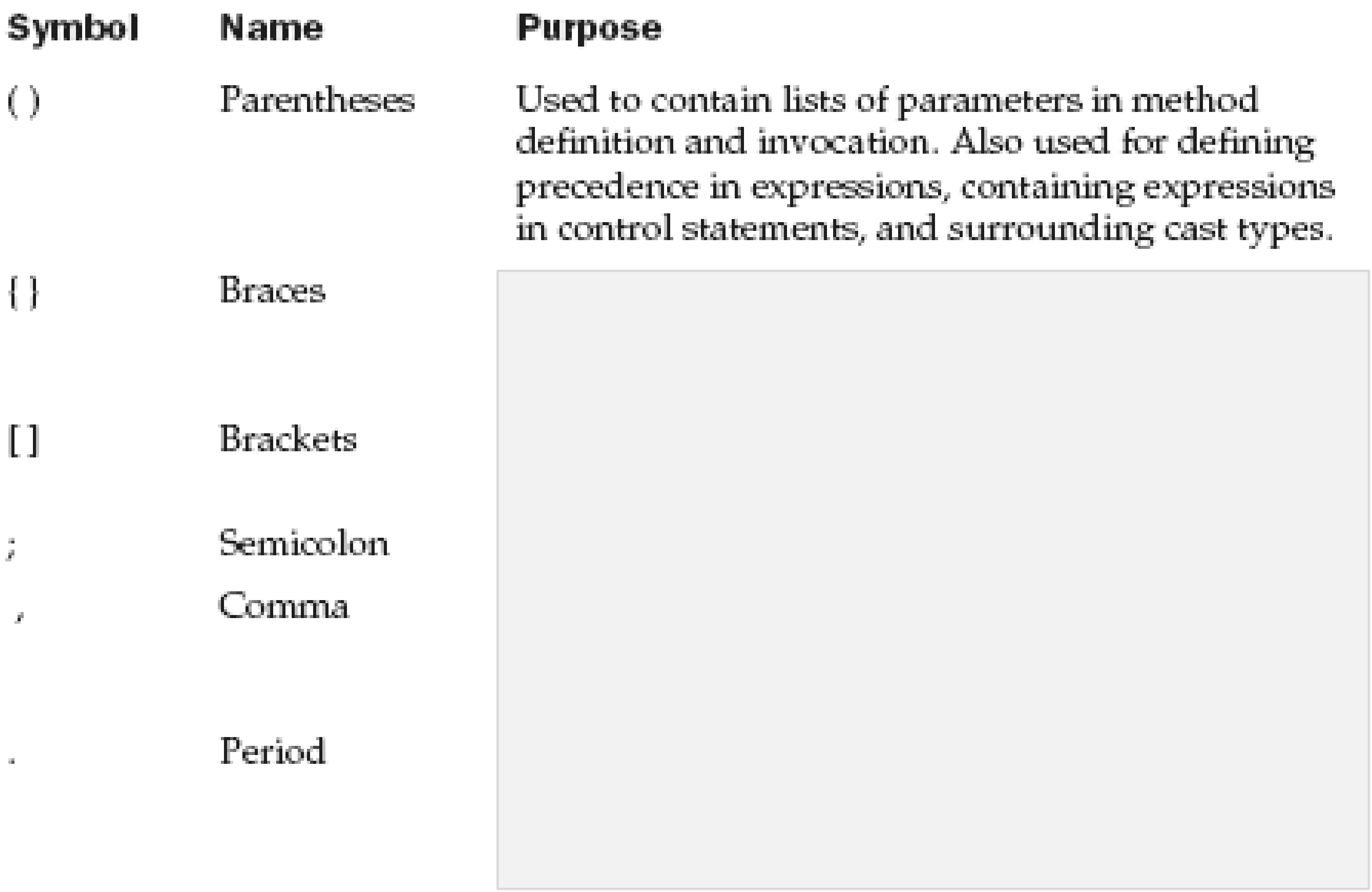
```
//   Java single-line comment

/*other Java comment style*/

/* also can go on
   for any number of lines*/
```

separators



Symbol	Name	Purpose
( )	Parentheses	
{ }	Braces	
[ ]	Brackets	
;	Semicolon	
,	Comma	
.	Period	

separators

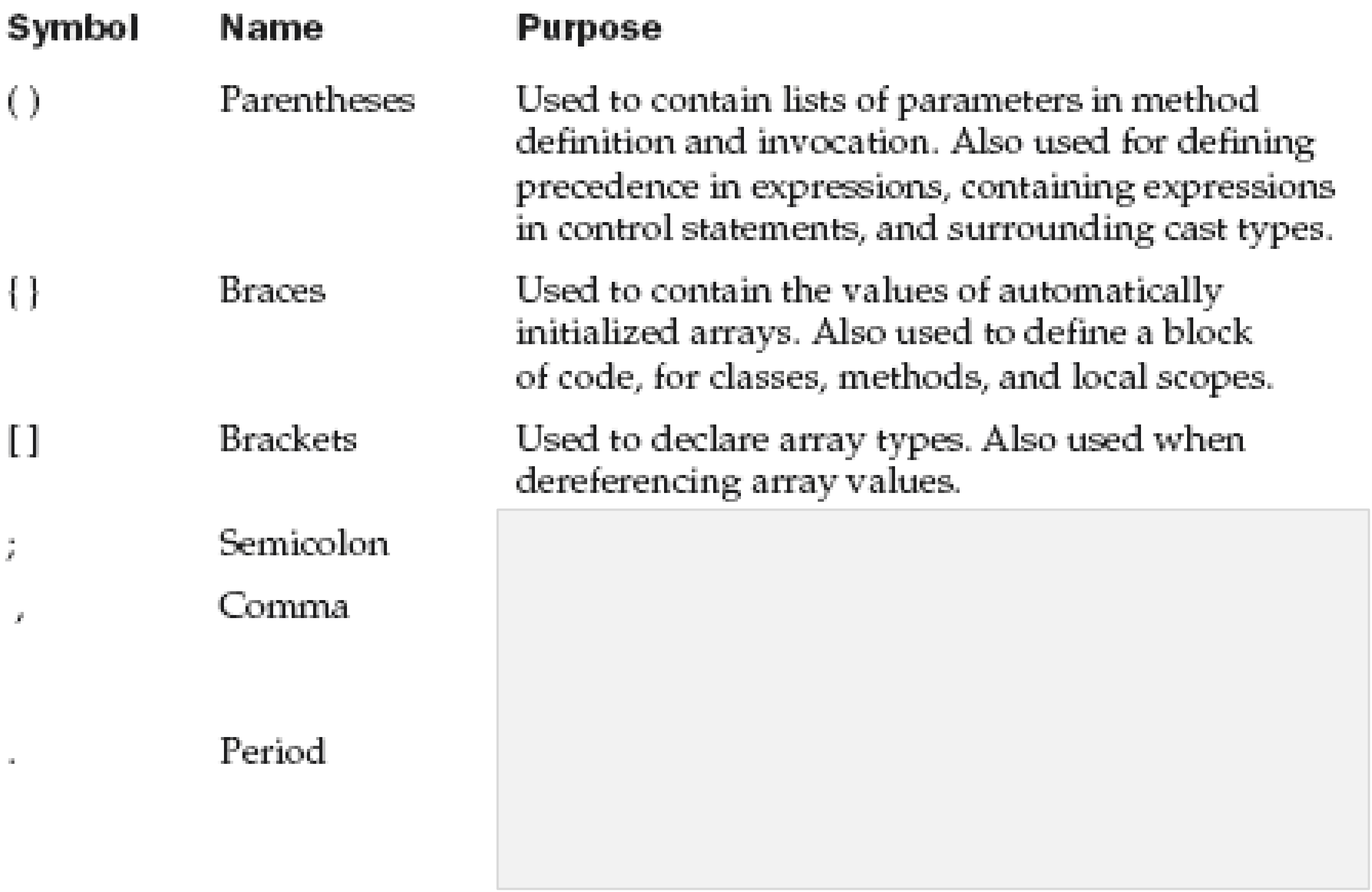


Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	
[ ]	Brackets	
;	Semicolon	
,	Comma	
.	Period	

separators

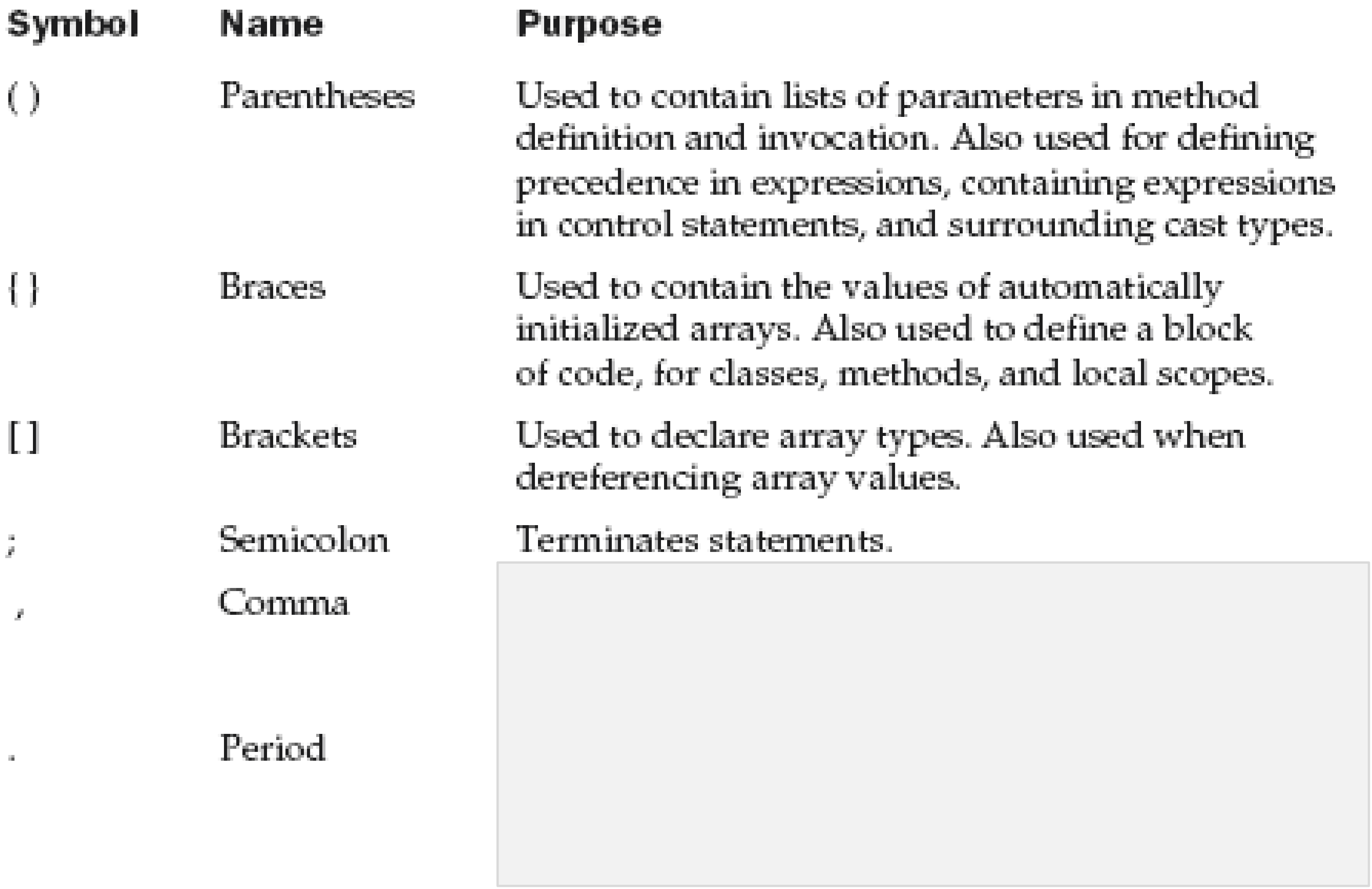
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	
;	Semicolon	
,	Comma	
.	Period	

separators



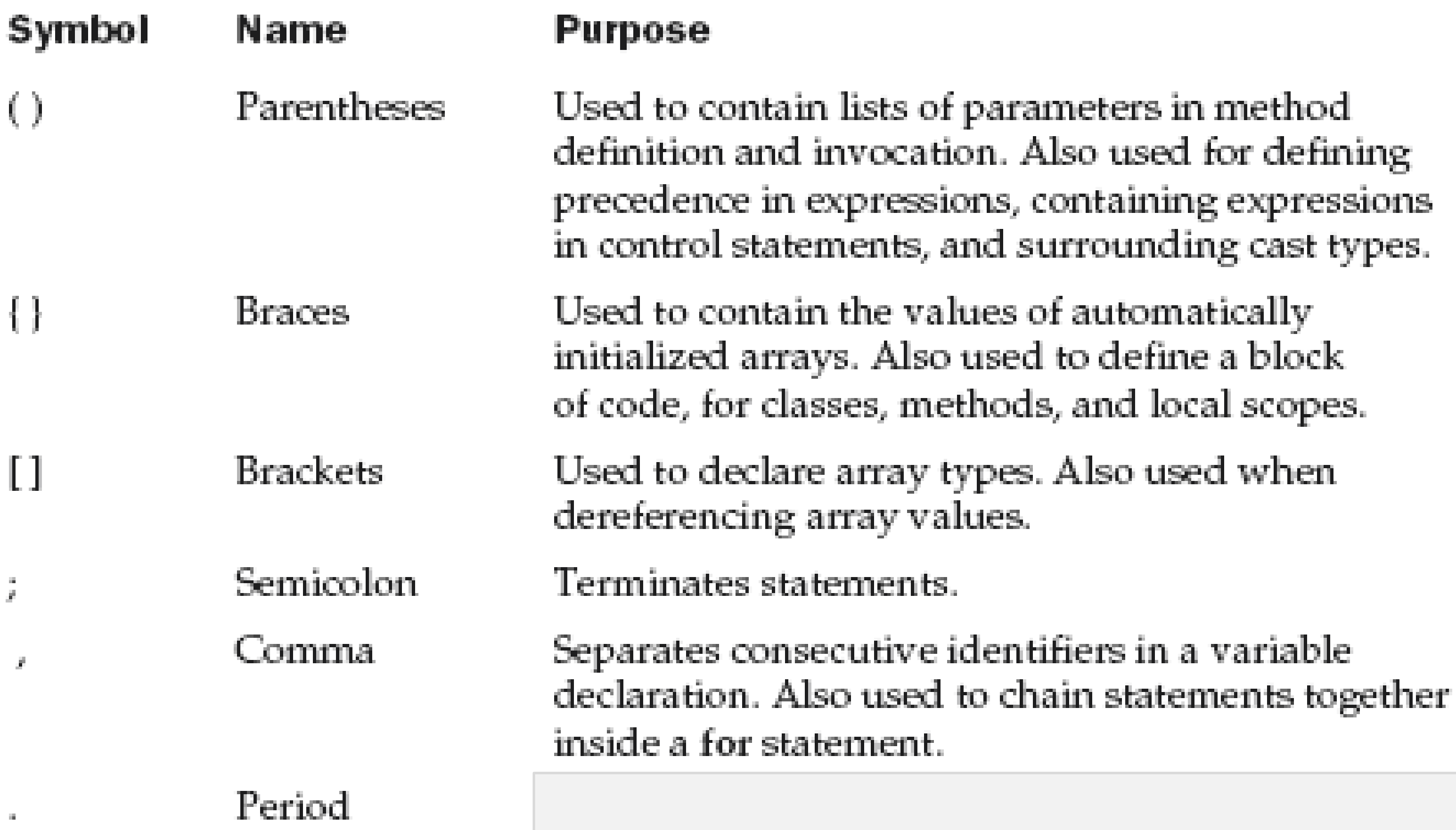
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	
,	Comma	
.	Period	

separators



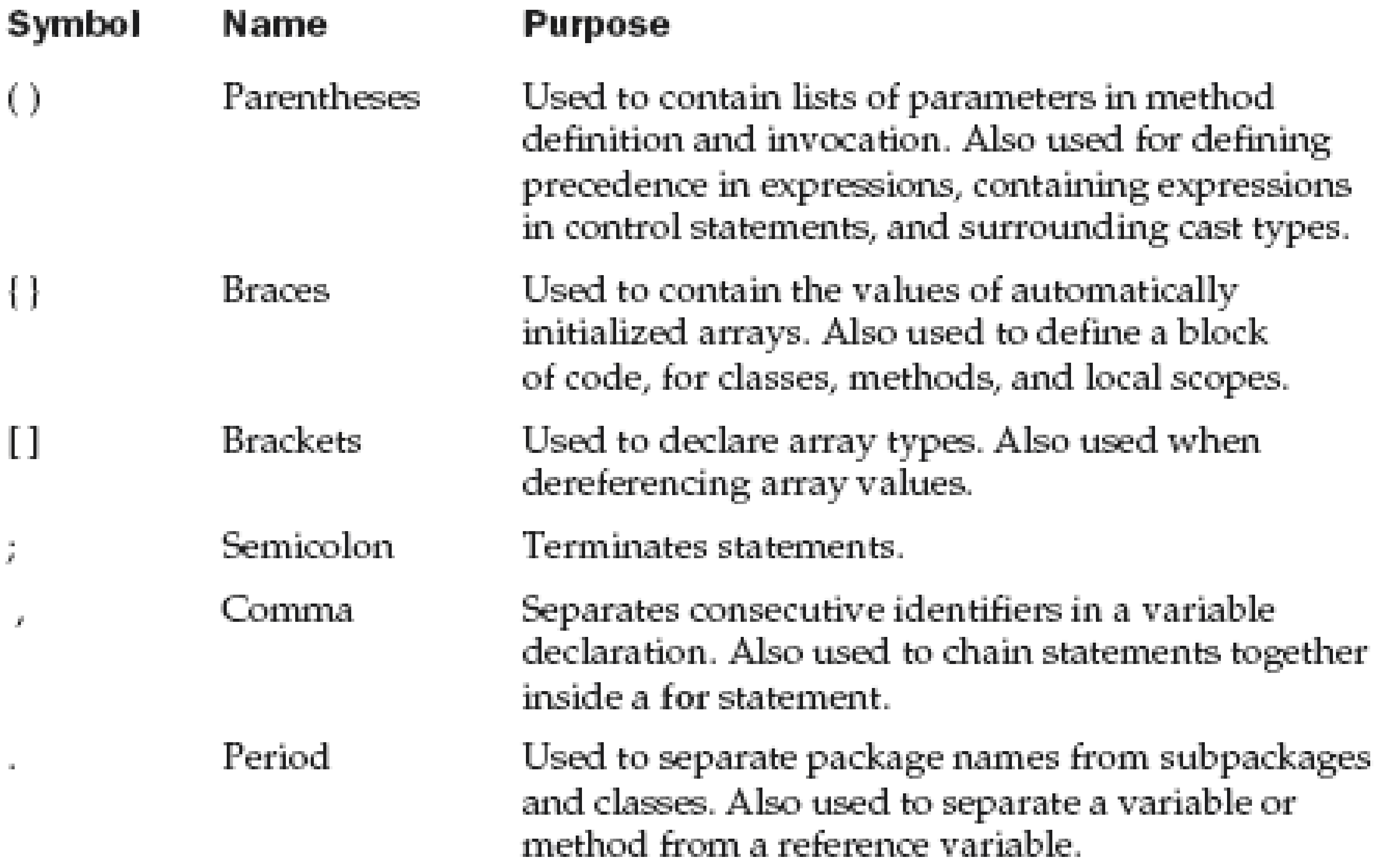
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	
.	Period	

separators



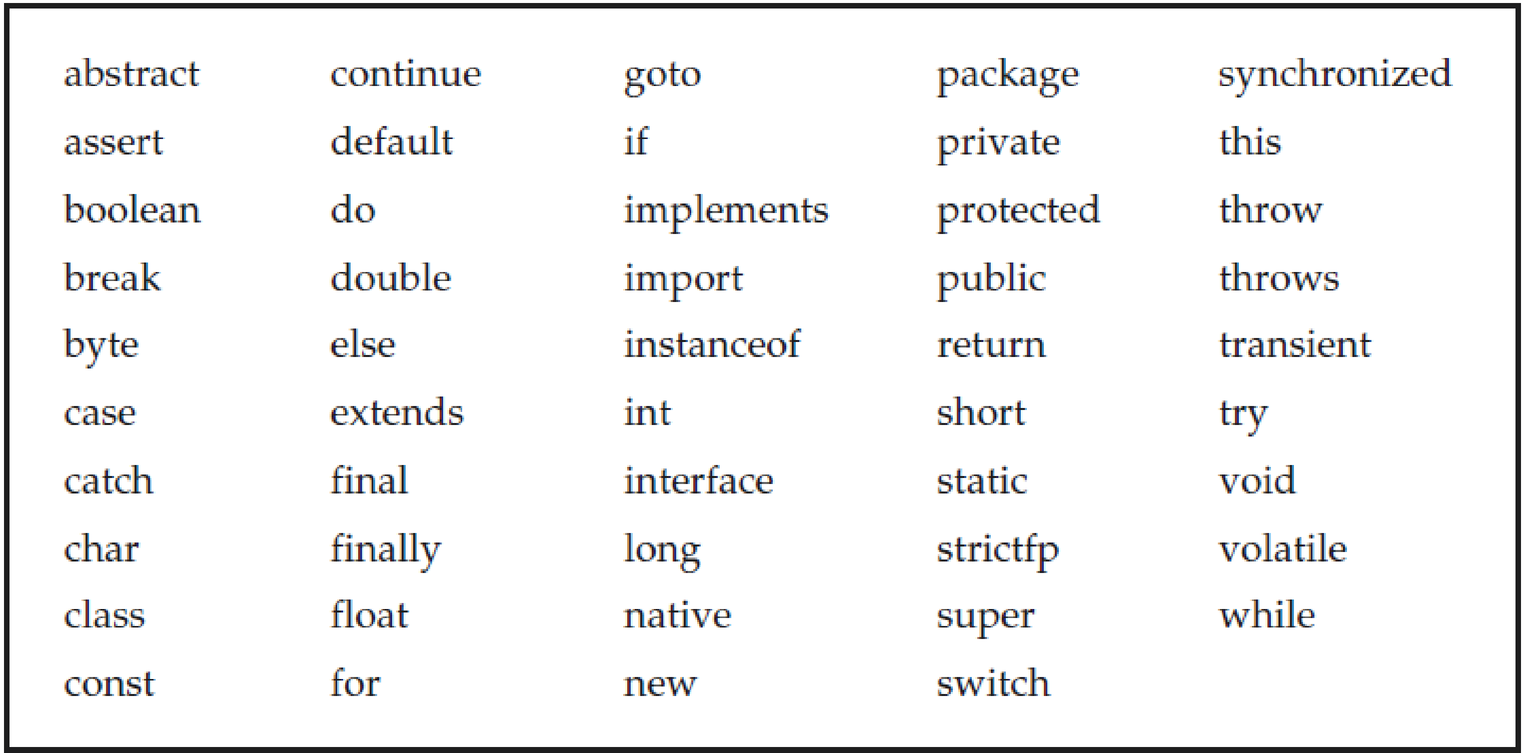
Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	

separators



Symbol	Name	Purpose
( )	Parentheses	Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types.
{ }	Braces	Used to contain the values of automatically initialized arrays. Also used to define a block of code, for classes, methods, and local scopes.
[ ]	Brackets	Used to declare array types. Also used when dereferencing array values.
;	Semicolon	Terminates statements.
,	Comma	Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside a for statement.
.	Period	Used to separate package names from subpackages and classes. Also used to separate a variable or method from a reference variable.

keywords



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

Table 2-1. *Java Reserved Keywords*

The Simple Types

Java defines eight simple types of data: **byte, short, int, long, char, float, double and boolean.**

These can be put in four groups:

Integers This group includes **byte, short, int**, and **long**, which are for whole valued signed numbers.

Floating-point numbers This group includes **float** and **double**, which represent numbers with fractional precision.

Characters This group includes **char**, which represents symbols in a character set, like letters and numbers.

Boolean This group includes **boolean**, which is a special type for representing true/false values.

The Simple Types

Java defines eight simple types of data: **byte, short, int, long, char, float, double and boolean.**

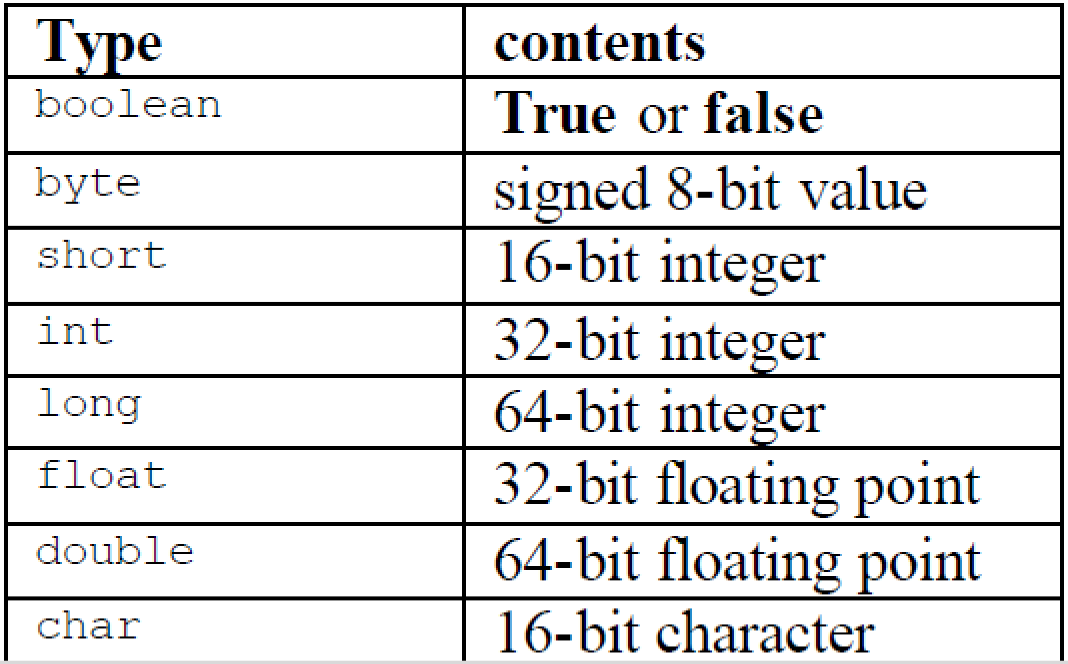
These can be put in four groups:

Integers This group includes **byte, short, int**, and **long**, which are for whole valued signed numbers.

Floating-point numbers This group includes **float** and **double**, which represent numbers with fractional precision.

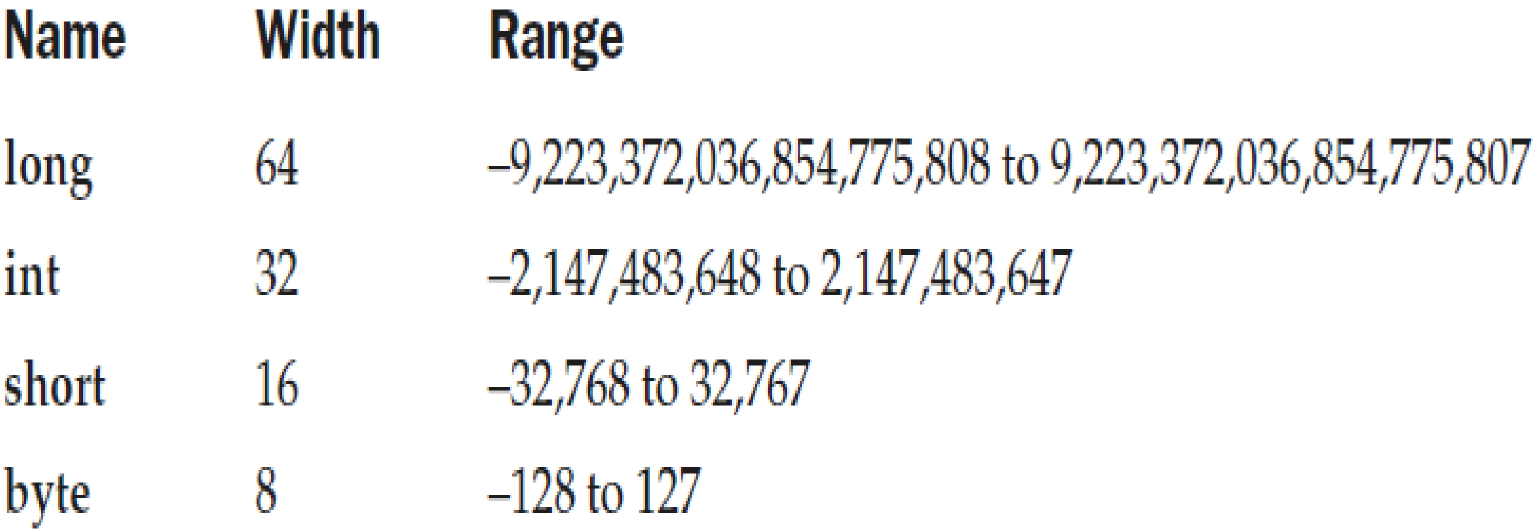
Characters This group includes **char**, which represents symbols in a character set, like letters and numbers.

Boolean This group includes **boolean**, which is a special type for representing true/false values.



Type	contents
`boolean`	**True** or **false**
`byte`	signed 8-bit value
`short`	16-bit integer
`int`	32-bit integer
`long`	64-bit integer
`float`	32-bit floating point
`double`	64-bit floating point
`char`	16-bit character

Integer Type

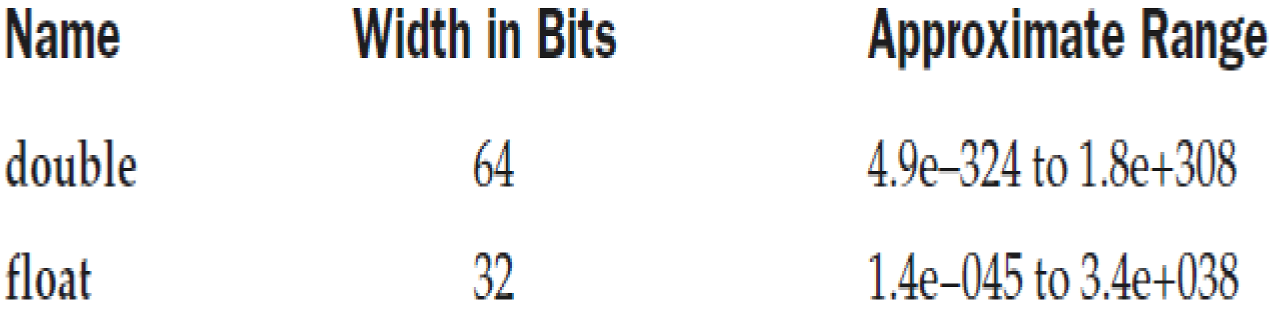


Name	Width	Range
long	64	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
int	32	–2,147,483,648 to 2,147,483,647
short	16	–32,768 to 32,767
byte	8	–128 to 127

Example

```
// Compute distance light travels using long variables.
class Light
{          public static void main(String args[])
           {
                  int lightspeed;
                  long days;
                  long seconds;
                  long distance;
                  // approximate speed of light in miles per second
                   lightspeed = 186000;
                   days = 1000;
                   seconds = days * 24 * 60 * 60;
                   distance = lightspeed * seconds;
                   System.out.print("In " + days+ ” days light will travel about”+ distance+
    “miles”);
   }
 }
```

Floating- Point Type



Name	Width in Bits	Approximate Range
double	64	4.9e–324 to 1.8e+308
float	32	1.4e–045 to 3.4e+038

Example

```
// Compute the area of a circle.
class Area
{
        public static void main(String args[])
       {
                       double  pi, r, a;
                       r = 10.8;
                      pi = 3.1416;
                       a = pi * r * r;
                      System.out.println("Area of circle is " + a);
          }
}
```

Character type

```
// Demonstrate char data type.
class CharDemo
{
   public static void main(String args[])
   {
           char ch1, ch2;
          ch1 = 88; // code for X
          ch2 = 'Y';
           System.out.print("ch1 and ch2: ");
          System.out.println(ch1 + " " + ch2);
  }
}
```

```
// char variables behave like integers.
class CharDemo2
{
          public static void main(String args[])
         {
                char ch1;
                ch1 = 'X';
               System.out.println("ch1 contains " + ch1);
               ch1++; // increment ch1
              System.out.println("ch1 is now " + ch1);
         }
}
```

Boolean

```
class BoolTest
{       public static void main(String args[])
          {       boolean b;
                b = false;
                System.out.println("b is " + b);
                b = true;
                System.out.println("b is " + b);
                if(b) System.out.println("This is executed.");
                b = false;
                if(b) System.out.println("This is not executed.");
                // outcome of a relational operator is a boolean value
                System.out.println("10 > 9 is " + (10 > 9));
      } }
```

Literals

Integer Literals

Floating point literals

Boolean literals

Character literals

String literals

Integer literals

decimal : example 4, 7, 89

Octal:

denoted by leading zeros

Example: 02 , 07

error : 09

Hexadecimal:

denoted by leading 0x or 0X

The range of a hexadecimal digit is 0 to 15, so A through F (or a through f) are

substituted for 10 through 15.

Floating- Point Literals

represent decimal values with a fractional component.

expressed in either **standard** or **scientific** notation.

Standard notation : consists of a whole number component followed by a decimal point followed by a fractional component.

For example, 2.0, 3.14159, and 0.6667

Floating- Point Literals

Scientific notation : uses a standard-notation, floating-point number plus a suffix that

specifies a power of 10 by which the number is to be multiplied.

The exponent is indicated by an E or e followed by a decimal number, which can be positive or negative.

Examples include 6.022E23, 12346E05

Boolean literals

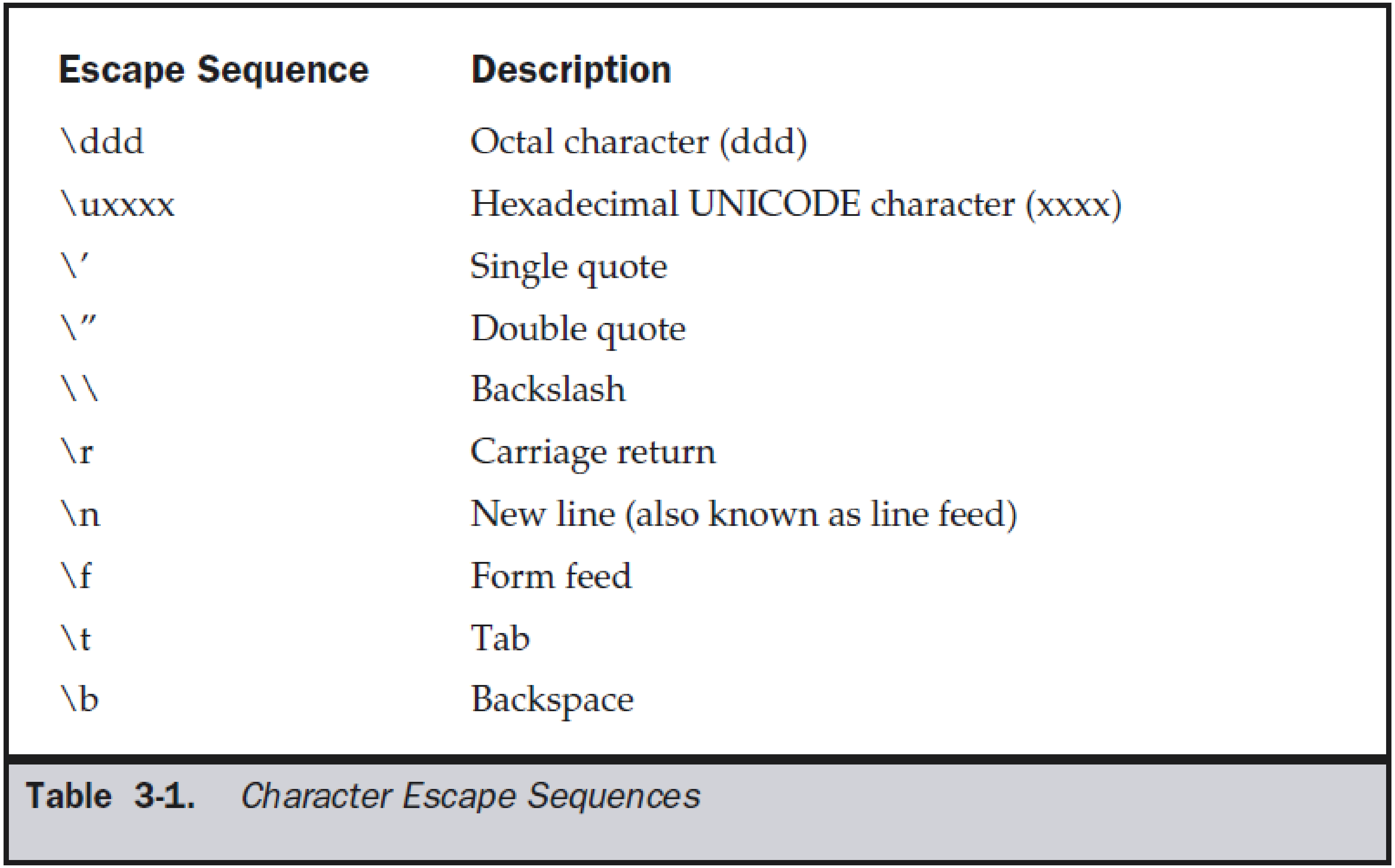
Boolean literals are simple.

There are only two logical values that a boolean value can have, true and false.

The values of true and false do not convert into any numerical representation.

The true literal in Java does not equal 1, nor does the false literal equal 0.

Character Literals



Escape Sequence	Description
\ddd	Octal character (ddd)
\uxxxx	Hexadecimal UNICODE character (xxxx)
\’	Single quote
\”	Double quote
\\	Backslash
\r	Carriage return
\n	New line (also known as line feed)
\f	Form feed
\t	Tab
\b	Backspace

Table 3-1. *Character Escape Sequences*

String literals

String literals are specified by enclosing a sequence of characters between a pair of double quotes.

Examples of string literals are

“Hello World”

“two\nlines”

“\”This is in quotes\””

Variables

Declaring a Variable

```
type identifier [ = value][, identifier [= value] ...] ;

int a, b, c;
int d = 3, e, f = 5;

byte z = 22;
double pi = 3.14159;

char x = 'x';
```

Dynamic Initialization

```
// Demonstrate dynamic initialization.
class DynInit
 {
           public static void main(String args[])
          {
                  double a = 3.0, b = 4.0;
                  // c is dynamically initialized
                   double c = Math.sqrt(a * a + b * b);
                  System.out.println("Hypotenuse is " + c);
           }
}
```

Scope of a Variable

- Scope
  - Part of program where a variable may be referenced
  - Determined by location of variable declaration
    - Boundary usually demarcated by { }
- Example

```
public MyMethod1() {
        int myVar;
        …
}
```

myVar accessible in method between { }

The Scope and Lifetime of Variables

variables declared inside a scope are not visible (that is, accessible) to code that is defined outside that scope

- Line 1: class Scope{
- Line 2:   Public static void main(String args[])
- Line 2a: {
- Line 3:   int x;
- Line 4:   x=100;
- Line 5:   if(x>=50)   {
- Line 6:   int y =200;
- Line 7:   System.out.println(“x”+x+”Y”+y);
- Line 8:   x=1000; }
- Line 9:   System.out.println(“x”+x);
- Line 10:   System.out.println(“Y”+y);
- Line 11:   }
- Line 12:   }

Operators

Operators

- Operators are special symbols used for:
  - mathematical functions
  - assignment statements
  - logical comparisons
- Examples of operators:
  - 3 + 5 // uses + operator
  - 14 + 5 – 4 * (5 – 3) // uses +, -, * operators
- Expressions: can be combinations of variables and operators that result in a value

Groups of Operators

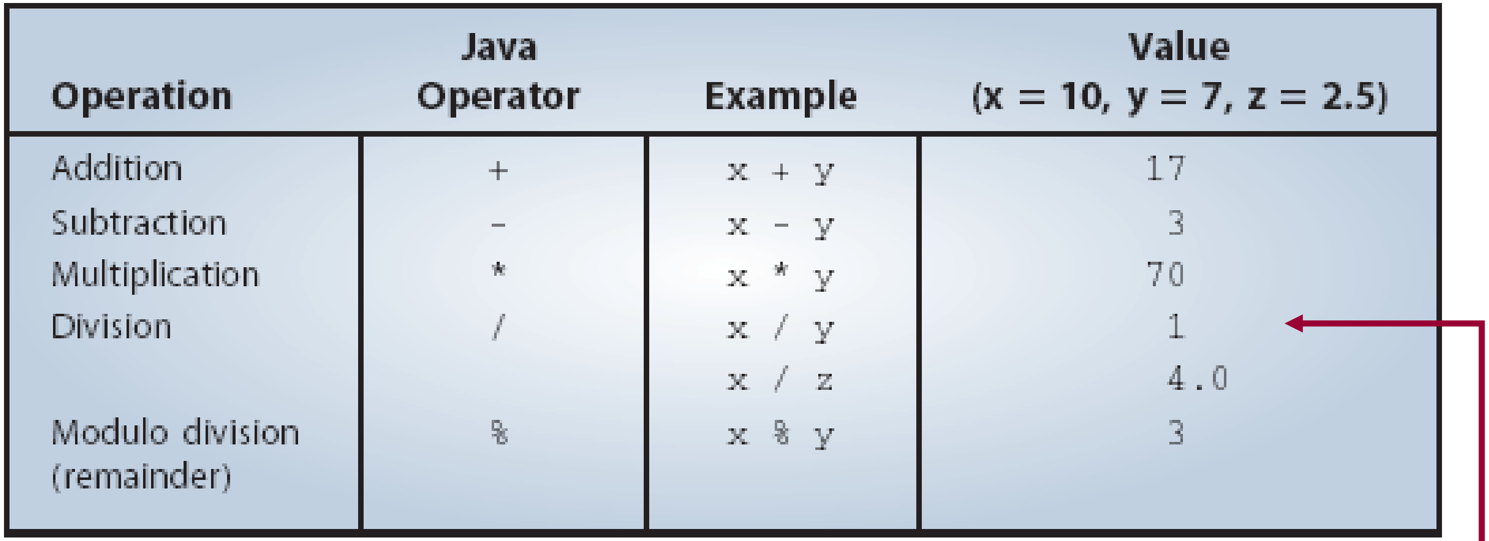
- There are 5 different groups of operators:
  - Arithmetic Operators
  - Assignment Operator
  - Increment / Decrement Operators
  - Relational Operators
  - Logical Operators

Java Arithmetic Operators

Addition	+
Subtraction	–
Multiplication	*
Division	/
Remainder (modulus)	%

Arithmetic Operators

- The following table summarizes the arithmetic operators available in Java.



Operation	Java Operator	Example	Value (x = 10, y = 7, z = 2.5)
Addition	+	x + y	17
Subtraction	-	x - y	3
Multiplication	*	x * y	70
Division	/	x / y	1
		x / z	4.0
Modulo division (remainder)	%	x % y	3

This is an integer division where the fractional part is truncated.

Example

Example of division issues:

10 / 3 gives 3

10.0 / 3 gives 3.33333

As we can see,

- if we divide two integers we get an integer result.
- if one or both operands is a floating-point value we get a floating-point result.

Modulus

- Generates the remainder when you divide two integer values.

5%3 gives 2   5%4 gives 1

5%5 gives 0   5%10 gives 5

- Modulus operator is most commonly used with integer operands. If we attempt to use the modulus operator on floating-point values we will get garbage!

Increment/Decrement Operators

Only use ++ or – – when a variable is being incremented/decremented as a statement by itself.

`x++;` is equivalent to `x = x+1;`

`x--;` is equivalent to `x = x-1;`

Relational Operators

- Relational operators compare two values
- They Produce a *boolean* value (**true** or **false)** depending on the relationship

Operation	**Is true when**
a >b	**a** is greater than **b**
a >=b	**a** is greater than or equal to **b**
a ==b	a is equal to **b**
a !=b	**a** is not equal to **b**
a <=b	**a** is less than or equal to **b**
a <b	**a** is less than **b**

Example

- int x = 3;
- int y = 5;
- boolean result;

  result = (x > y);
- now result is assigned the value false because 3 is not greater than 5

Relational Operator Examples

```
public class Example {
   public static void main(String[] args) {
       int p =2; int q = 2; int r = 3;
       System.out.println("p < r " + (p < r));
       System.out.println("p > r " + (p > r));
       System.out.println("p == q " + (p == q));
       System.out.println("p != q " + (p != q));

              }
}
```

```
> java Example
p < r true
p > r false
p == q true
p != q false

>
```

Logical Operators (boolean)

&& || !

<table>
<tr><td>Logical AND</td><td>&</td></tr>
<tr><td>Logical OR</td><td>|</td></tr>
<tr><td>Logical NOT</td><td>!</td></tr>
<tr><td>Short-circuit OR</td><td>||</td></tr>
<tr><td>Short-Circuit AND</td><td>&&</td></tr>
</table>

Logical (&&) Operator Examples

```
public class Example {
   public static void main(String[] args) {
      boolean t = true;
      boolean f = false;

      System.out.println("f & f " + (f & f));
      System.out.println("f & t " + (f & t));
      System.out.println("t & f " + (t & f));
      System.out.println("t & t " + (t & t));

   }
}
```

```
> java Example
f & f false
f & t false
t & f false
t & t true
>
```

Logical (||) Operator Examples

```
public class Example {
   public static void main(String[] args) {
      boolean t = true;
      boolean f = false;

      System.out.println("f | f " + (f | f));
      System.out.println("f | t " + (f | t));
      System.out.println("t | f " + (t | f));
      System.out.println("t | t " + (t | t));

   }
}
```

```
> java Example
f | f false
f | t true
t | f true
t | t true
>
```

Logical (!) Operator Examples

```
public class Example {
  public static void main(String[] args) {
      boolean t = true;
      boolean f = false;

      System.out.println("!f " + !f);
      System.out.println("!t " + !t);

  }
}
```

```
> java Example
!f true
!t false
>
```

Shift Operators (Bit Level)

<< >> >>>

- **Shift Left << Fill with Zeros**

For each shift left, the high-order bit is shifted out (and lost), and a zero is brought in on the right.

- **Shift Right >> Based on Sign**

When shifting right, the top (leftmost) bits exposed by the right shift are filled in with the previous contents of the top bit.

This is called *sign extension and serves* to preserve the sign of negative numbers when we shift them right.

- **Shift Right >>> Fill with Zeros**

Logical Operator Examples

Short Circuiting with &&

```
public class Example {
   public static void main(String[] args) {
      boolean b;
      int j, k;

      j = 0; k = 0;
      b = ( j++ == k ) && ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);

      j = 0; k = 0;
      b = ( j++ != k ) && ( j == ++k );
      System.out.println("b, j, k " + b + ", " + j + ", " + k);
   }
}
```

```
> java Example
b, j, k true 1, 1
b, j, k false 1, 0
>
```

Logical Operator Examples

Short Circuiting with ||

```
public class Example {
   public static void main(String[] args) {
        boolean b;
        int j, k;

        j = 0; k = 0;
        b = ( j++ == k ) || ( j == ++k );
        System.out.println("b, j, k " + b + ", " + j + ", " + k);

        j = 0; k = 0;
        b = ( j++ != k ) || ( j == ++k );
        System.out.println("b, j, k " + b + ", " + j + ", " + k);
   }
}
```

```
> java Example
b, j, k true 1, 0
b, j, k true 1, 1
>
```

Logical Operators (Bit Level)

& | ^ ~

AND	&
OR	|
XOR	^
NOT	~

Twos Complement Numbers

Base 10	A byte of binary
+127	01111111
+4	00000100
+3	00000011
+2	00000010
+1	00000001
+0	00000000
-1	11111111
-2	11111110
-3	11111101
-4	11111100
-128	10000000

Adding Twos Complements

Base 10	Binary
+3	00000011
-2	11111110
+1	00000001

Base 10	Binary
+2	00000010
-3	11111101
-1	11111111

Logical Operators (Bit Level)

& | ^ ~

```
int a = 10; // 00001010 = 10
int b = 12; // 00001100 = 12
```

&

AND

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a & b   00000000000000000000000000001000     8
```

|

OR

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a | b   00000000000000000000000000001110    14
```

^

XOR

```
a       00000000000000000000000000001010    10
b       00000000000000000000000000001100    12
a ^ b   00000000000000000000000000000110     6
```

~

NOT

```
 a      00000000000000000000000000001010    10
~a      11111111111111111111111111110101   -11
```

The value in binary is:value1 =
00000000000000000000000000001010
By applying the complement operator,:
11111111111111111111111111110101
But the result displayed will be:-11

This is the one’s complement of the decimal number 10 And since the first (leftmost) bit is 1 in binary, it means that the sign is negative for the number that is stored.
Now, since the numbers are stored as 2’s complement, first we need to find its 2’s complement and then convert the resultant binary number into a decimal number so the result becomes -11

Logical (bit) Operator Examples

```
public class Example {
  public static void main(String[] args) {
      int a = 10;     // 00001010 = 10
      int b = 12;     // 00001100 = 12
      int and, or, xor, na;
      and = a & b;     // 00001000 = 8
      or  = a | b;     // 00001110 = 14
      xor = a ^ b;     // 00000110 = 6
      na  = ~a;        // 11110101 = -11
      System.out.println("and " + and);
      System.out.println("or " + or);
      System.out.println("xor " + xor);
      System.out.println("na " + na);
  }
}
```

```
> java Example
and 8
or 14
xor 6
na -11
>
```

Shift Operators << >>

```
int a =  3; // ...00000011 =  3
int b = -4; // ...11111100 = -4
```

<<

Left

```
a        00000000000000000000000000000011     3
a << 2   00000000000000000000000000001100    12

b        11111111111111111111111111111100    -4
b << 2   11111111111111111111111111110000   -16
```

>>

Right

```
a        00000000000000000000000000000011     3
a >> 2   00000000000000000000000000000000     0

b        11111111111111111111111111111100    -4
b >> 2   11111111111111111111111111111111    -1
```

Shift Operator >>>

```
int a =  3; // ...00000011 =  3
int b = -4; // ...11111100 = -4
```

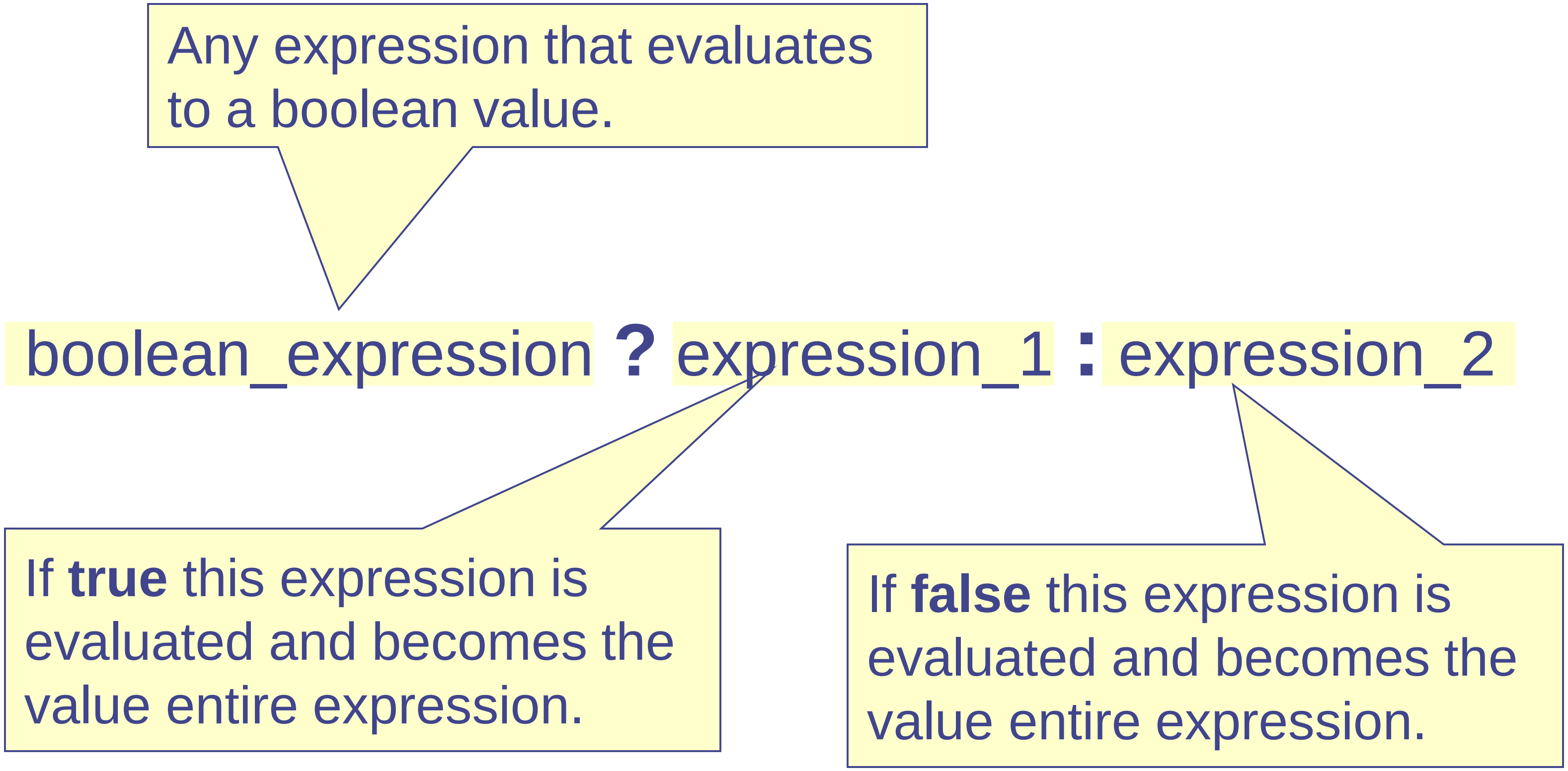
>>>

Right 0

```
a        00000000000000000000000000000011      3
a >>> 2  00000000000000000000000000000000      0

b        11111111111111111111111111111100     -4
b >>> 2  00111111111111111111111111111111     +big
```

Ternary Operator

?   :



Any expression that evaluates to a boolean value.

boolean_expression **?** expression_1 **:** expression_2

If **true** this expression is evaluated and becomes the value entire expression.

If **false** this expression is evaluated and becomes the value entire expression.

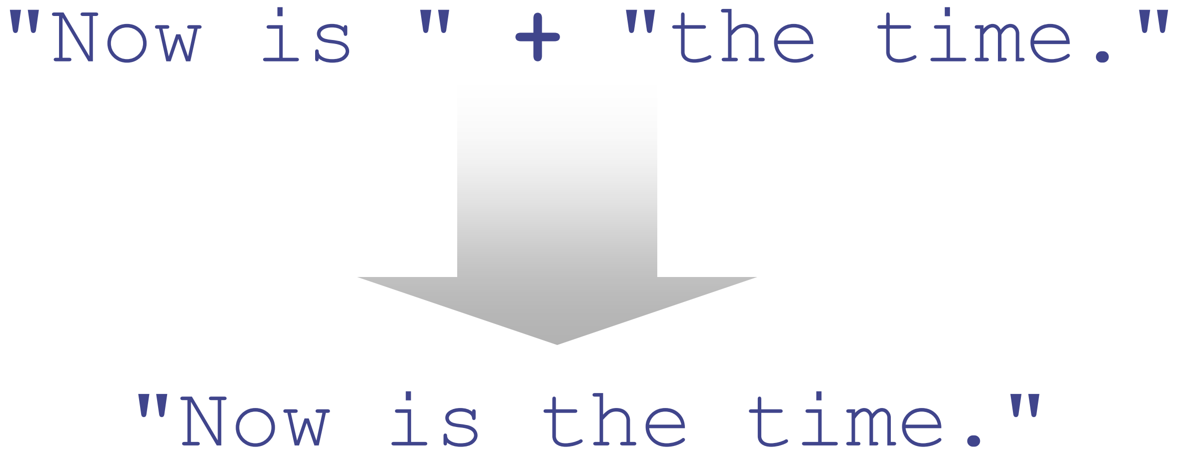
Examples

```
public class Example {
   public static void main(String[] args) {
      boolean t = true;
      boolean f = false;

      System.out.println("t?true:false "+(t ? true : false ));
      System.out.println("t?1:2 "+(t ? 1 : 2 ));
      System.out.println("f?true:false "+(f ? true : false ));
      System.out.println("f?1:2 "+(f ? 1 : 2 ));
   }
}
```

```
> java Example
t?true:false true
t?1:2 1
f?true:false false
f?1:2 2
>
```

String (+) Operator

String Concatenation



```
"Now is " + "the time."
```

```
"Now is the time."
```

String (+) Operator

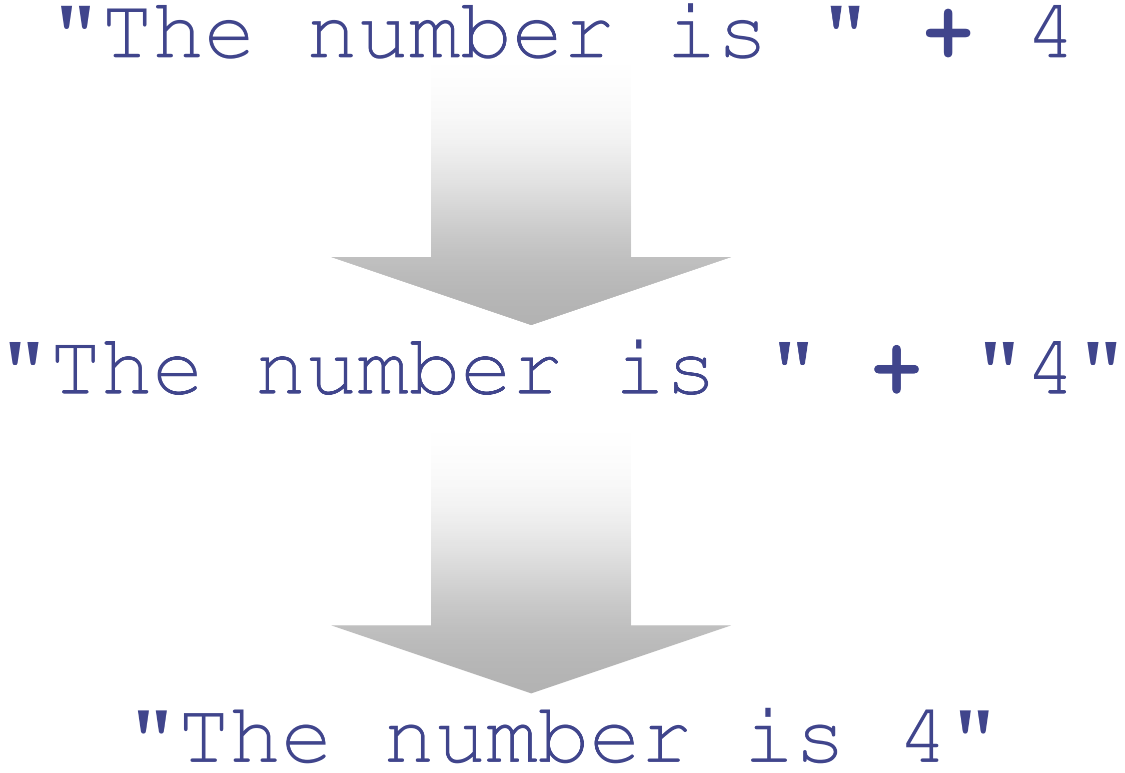
Automatic Conversion to a String

expression_1 **+** expression_2

If either **expression_1** or **expression_2** evaluates to a string the other will be converted to a string if needed. The result will be their concatenation.

String (+) Operator

Automatic Conversion with Primitives



```
"The number is " + 4
```

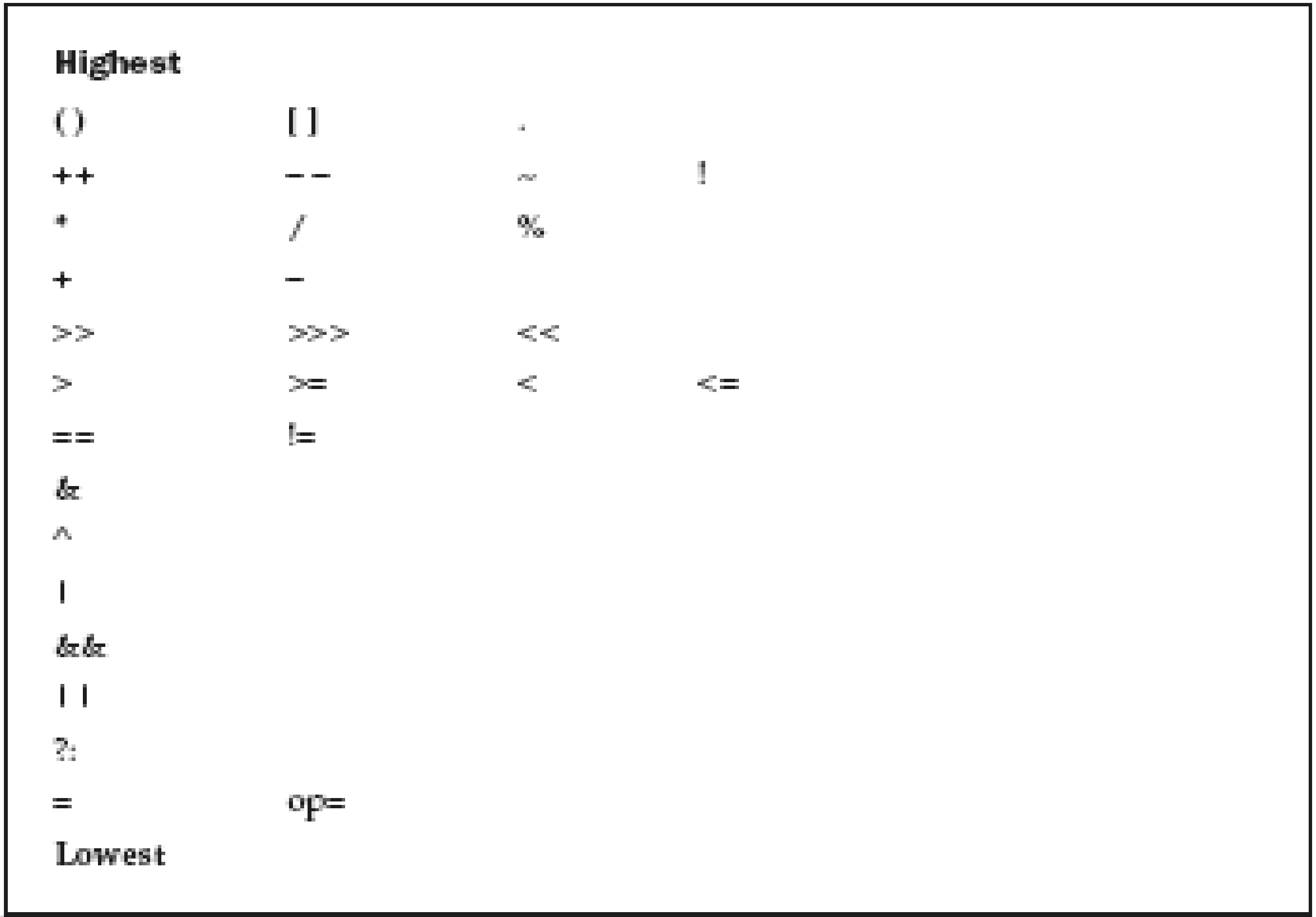
```
"The number is " + "4"
```

```
"The number is 4"
```

Operators Precedence

Parentheses	(), inside-out
Increment/decrement	++, --, from left to right
Multiplicative	*, /, %, from left to right
Additive	+, -, from left to right
Relational	<, >, <=, >=, from left to right
Equality	==, !=, from left to right
Logical AND	&&
Logical OR	||
Assignment	=, +=, -=, *=, /=, %=

Operator Precedence



Highest			
( )	[ ]	.	
++	– –	~	!
*	/	%	
+	–		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			
&&			
| |			
?:			
=	op=		
Lowest			

Table 4-1. *The Precedence of the Java Operators*

```
int a=128, b=2, x;

x = a >> b + 3;

System.out.println(“x=“ +x);
```

```
int a=128, b=2, x;

x = a >> b + 3;

System.out.println(“x=“ +x);
```

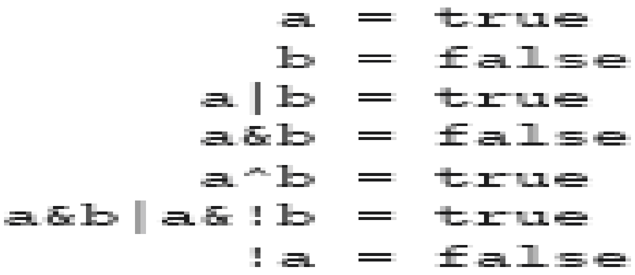
```
X = 4
```

Practice Code Snippets

Code Snippet 1

```
  boolean a = true,  b = false;
boolean c = a | b;
boolean d = a & b;
boolean e = a ^ b;
boolean f = (!a & b) | (a & !b);
boolean g = !a;
System.out.println(" a = " + a);
System.out.println(" b = " + b);
System.out.println(" a|b = " + c);
System.out.println(" a&b = " + d);
System.out.println(" a^b = " + e);
System.out.println("!a&b|a&!b = " + f);
System.out.println(" !a = " + g);
```

Answer



```
       a = true
       b = false
     a|b = true
     a&b = false
     a^b = true
a&b|a&!b = true
      !a = false
```

Code Snippet 2

```
int a=5,b=6;
if(a >1 | ++b > 1)
{
        System.out.println("b="+b);
}

int a=5,b=6;
if(a >1 || ++b > 1)
{
        System.out.println("b="+b);
}
```

Code Snippet 3

```
 int i, k;
i = 10;

k = i < 0 ? -i : i;        // get absolute value of i
System.out.print("Absolute value of “+ i + " is " + k);

  i = -10;
k = i < 0 ? -i : i;       // get absolute value of i
System.out.print("Absolute value of “+ i + " is " + k);
```

Code Snippet 4

```
int a=128, b=2, x;

x = a >> b + 3;

System.out.println(“x=“ +x);
```

X = 4