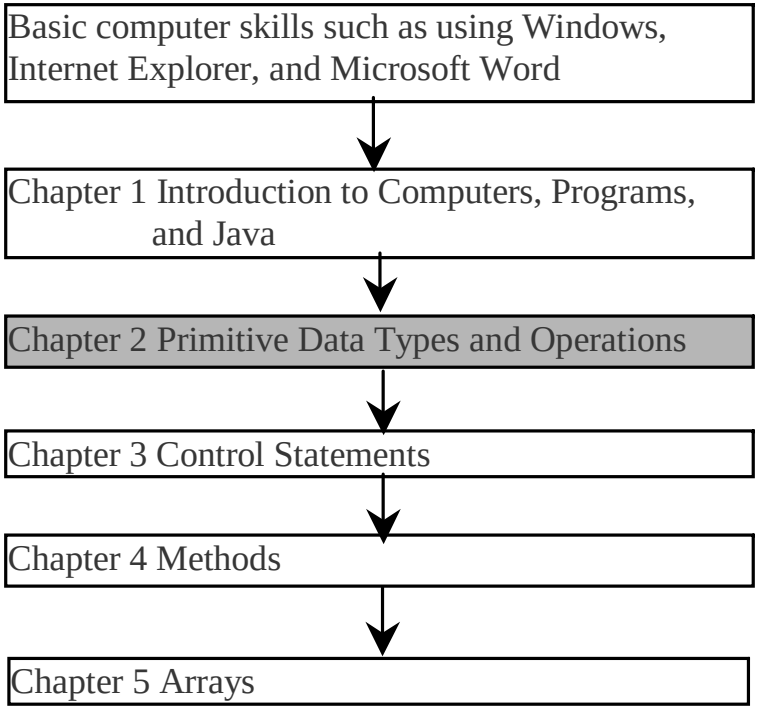
Chapter 2 Primitive Data Types and Operations

Prerequisites for Part I



Objectives

- To write Java programs to perform simple calculations (§2.2).
- To use identifiers to name variables, constants, methods, and classes (§2.3).
- To use variables to store data (§2.4-2.5).
- To program with assignment statements and assignment expressions (§2.5).
- To use constants to store permanent data (§2.6).
- To declare Java primitive data types: byte, short, int, long, float, double, char, and boolean (§2.7 – 2.10).
- To use Java operators to write expressions (§2.7 – 2.10).
- To know the rules governing operand evaluation order, operator precedence, and operator associativity (§2.11 – 2.12).
- To represent a string using the String type. (§2.13)
- To obtain input from console (§2.16 Optional).
- To format output using JDK 1.5 printf (§2.17).
- To become familiar with Java documentation, programming style, and naming conventions (§2.18).
- To distinguish syntax errors, runtime errors, and logic errors (§2.19).

Introducing Programming with an Example

Example 2.1 Computing the Area of a Circle

This program computes the area of the circle.

ComputeArea

Identifiers

- An identifier is a sequence of characters that consist of letters, digits, underscores (_), and dollar signs ($).
- An identifier must start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
  - An identifier cannot be a reserved word. (See Appendix A, "Java Keywords," for a list of reserved words).
- An identifier cannot be `true`, `false`, or `null`.
- An identifier can be of any length.

Variables

```
// Compute the first area
radius = 1.0;
area = radius * radius * 3.14159;
System.out.println("The area is “ +
  area + " for radius "+radius);

// Compute the second area
radius = 2.0;
area = radius * radius * 3.14159;
System.out.println("The area is “ +
  area + " for radius "+radius);
```

Declaring Variables

```
int x;          // Declare x to be an
                // integer variable;
double radius; // Declare radius to
                // be a double variable;
char a;         // Declare a to be a
                // character variable;
```

Assignment Statements

```
x = 1;              // Assign 1 to x;

radius = 1.0;       // Assign 1.0 to radius;

a = 'A';            // Assign 'A' to a;
```

Declaring and Initializing in One Step

- `int x = 1;`
- `double d = 1.4;`
- `float f = 1.4;`

  Is this statement correct?

Constants

```
final datatype CONSTANTNAME = VALUE;

final double PI = 3.14159;
final int SIZE = 3;
```

Numerical Data Types

`byte`	8 bits
`short`	16 bits
`int`	32 bits
`long`	64 bits
`float`	32 bits
`double`	64 bits

Operators

+, -, *, /, and %

5 / 2 yields an integer 2.

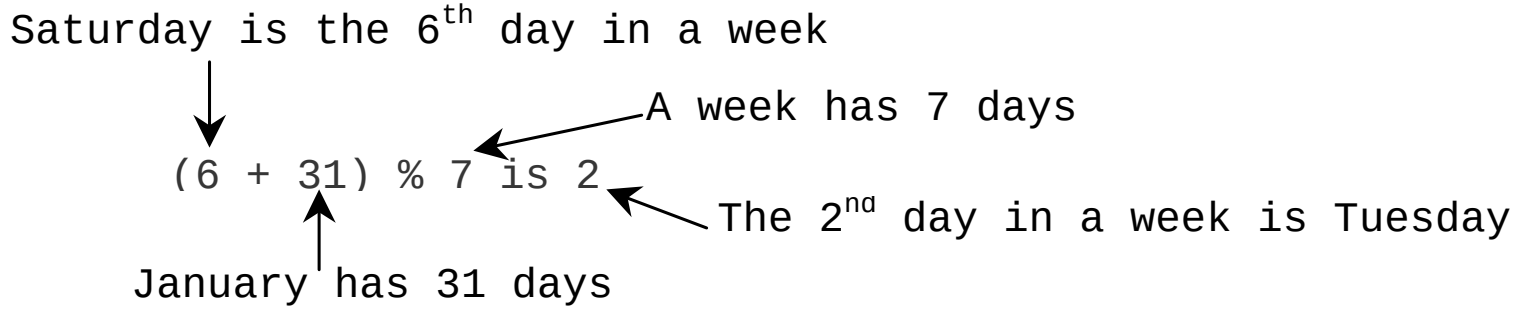
5.0 / 2 yields a double value 2.5

5 % 2 yields 1 (the remainder of the division)

Remainder Operator

- The % symbol is the remainder operator

- Suppose you know January 1, 2005 is Saturday, you can find that the day for February 1, 2005 is Tuesday using the following expression:



NOTE

Calculations involving floating-point numbers are approximated because these numbers are not stored with complete accuracy. For example,

System.out.println(1 - 0.1 - 0.1 - 0.1 - 0.1 - 0.1);

displays 0.5000000000000001, not 0.5, and

System.out.println(1.0 - 0.9);

displays 0.09999999999999998, not 0.1. Integers are stored precisely. Therefore, calculations with integers yield a precise integer result.

Number Literals

A *literal* is a constant value that appears directly in the program. For example, 34, 1,000,000, and 5.0 are literals in the following statements:

```
int i = 34;

long x = 1000000;

double d = 5.0;
```

Integer Literals

- An integer literal can be assigned to an integer variable as long as it can fit into the variable.

- A compilation error would occur if the literal were too large for the variable to hold. For example, the statement byte b = 1000 would cause a compilation error, because 1000 cannot be stored in a variable of the byte type.

- An integer literal is assumed to be of the int type, whose value is between -2^{31} (-2147483648) to $2^{31}-1$ (2147483647).

- To denote an integer literal of the long type, append it with the letter L or l. L is preferred because l (lowercase L) can easily be confused with 1 (the digit one).

Floating-Point Literals

- Floating-point literals are written with a decimal point. By default, a floating-point literal is treated as a double type value.
- For example, 5.0 is considered a double value, not a float value.
- You can make a number a float by appending the letter f or F, and make a number a double by appending the letter d or D.
- For example, you can use 100.2f or 100.2F for a float number, and 100.2d or 100.2D for a double number.

Scientific Notation

Floating-point literals can also be specified in scientific notation, for example, 1.23456e+2, same as 1.23456e2, is equivalent to 123.456, and 1.23456e-2 is equivalent to 0.0123456.

E (or e) represents an exponent and it can be either in lowercase or uppercase.

Arithmetic Expressions

$$\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

is translated to

```
(3+4*x)/5 – 10*(y-5)*(a+b+c)/x + 9*(4/x + (9+x)/y)
```

Shortcut Assignment Operators

Operator	*Example*	*Equivalent*
`+=`	`i+=8`	`i = i+8`
`-=`	`f-=8.0`	`f = f-8.0`
`*=`	`i*=8`	`i = i*8`
`/=`	`i/=8`	`i = i/8`
`%=`	`i%=8`	`i = i%8`

Increment and Decrement Operators

Operator	Name	Description
++var	preincrement	++var increments var by 1 and evaluates to the *new* value in var *after* the increment.
var++	postincrement	var++ evaluates to the *original* value in var and increments var by 1.
--var	predecrement	--var decrements var by 1 and evaluates to the *new* value in var *after* the decrement.
var--	postdecrement	var-- evaluates to the *original* value in var and decrements var by 1.

Increment and Decrement Operators, cont.

```
int i = 10;
int newNum = 10 * i++;
```

Same effect as

```
int newNum = 10 * i;
i = i + 1;
```

```
int i = 10;
int newNum = 10 * (++i);
```

Same effect as

```
i = i + 1;
int newNum = 10 * i;
```

Increment and Decrement Operators, cont.

Using increment and decrement operators makes expressions short, but it also makes them complex and difficult to read. Avoid using these operators in expressions that modify multiple variables, or the same variable for multiple times such as this: int k = ++i + i.

Assignment Expressions and Assignment Statements

Prior to Java 2, all the expressions can be used as statements. Since Java 2, only the following types of expressions can be statements:

variable op= expression; // Where op is +, -, *, /, or %

++variable;

variable++;

--variable;

variable--;

Numeric Type Conversion

Consider the following statements:

```
byte i = 100;
long k = i * 3 + 4;
double d = i * 3.1 + k / 2;
```

Conversion Rules

When performing a binary operation involving two operands of different types, Java automatically converts the operand based on the following rules:

1. If one of the operands is double, the other is converted into double.
2. Otherwise, if one of the operands is float, the other is converted into float.
3. Otherwise, if one of the operands is long, the other is converted into long.
4. Otherwise, both operands are converted into int.

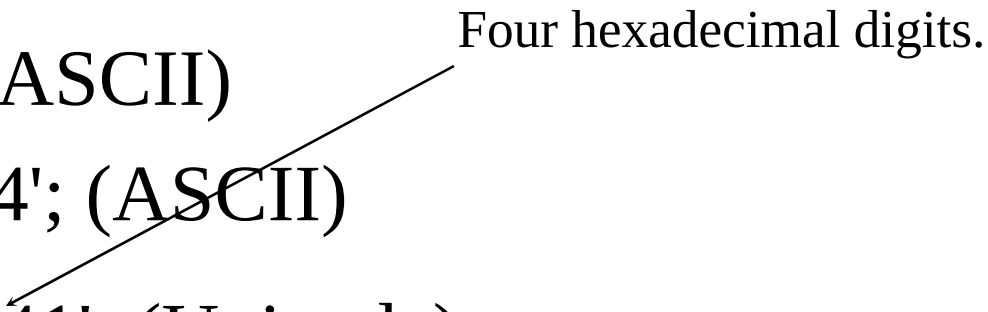
Type Casting

```
Implicit casting
  double d = 3; (type widening)

Explicit casting
  int i = (int)3.0; (type narrowing)
  int i = (int)3.9; (Fraction part
  is truncated)
```

What is wrong? int x = 5 / 2.0;

Character Data Type



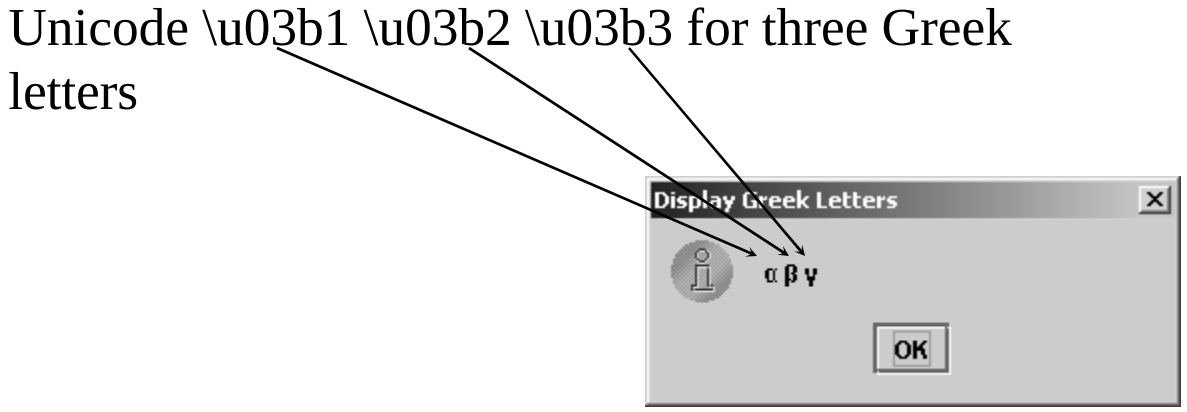

```
char letter = 'A'; (ASCII)
char numChar = '4'; (ASCII)
char letter = '\u0041'; (Unicode)
char numChar = '\u0034'; (Unicode)
```

NOTE: The increment and decrement operators can also be used on char variables to get the next or preceding Unicode character. For example, the following statements display character b.

```
char ch = 'a';
System.out.println(++ch);
```

Unicode Format

Java characters use *Unicode*, a 16-bit encoding scheme established by the Unicode Consortium to support the interchange, processing, and display of written texts in the world's diverse languages. Unicode takes two bytes, preceded by \u, expressed in four hexadecimal numbers that run from '\u0000' to '\uFFFF'. So, Unicode can represent `65535 + 1 characters`.



Escape Sequences for Special Characters

Description	*Escape Sequence*	*Unicode*
Backspace	`\b`	`\u0008`
Tab	`\t`	`\u0009`
Linefeed	`\n`	`\u000A`
Carriage return	`\r`	`\u000D`
Backslash	`\\`	`\u005C`
Single Quote	`\'`	`\u0027`
Double Quote	`\"`	`\u0022`

Appendix B: ASCII Character Set

ASCII Character Set is a subset of the Unicode from \u0000 to \u007f

TABLE B.1 ASCII Character Set in the Decimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht
1	nl	vt	ff	cr	so	si	dle	dcl	dc2	dc3
2	dc4	nak	syn	etb	can	em	sub	esc	fs	gs
3	rs	us	sp	!	″	#	$	%	&	’
4	(	)	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[	\	]	^	_	‘	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{	|	}	~	del		

ASCII Character Set, cont.

ASCII Character Set is a subset of the Unicode from \u0000 to \u007f

TABLE B.2 ASCII Character Set in the Hexadecimal Index

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*A*	*B*	*C*	*D*	*E*	*F*
0	nul	soh	stx	etx	eot	enq	ack	bel	bs	ht	nl	vt	ff	cr	so	si
1	dle	dcl	dc2	dc3	dc4	nak	syn	etb	can	em	sub	esc	fs	gs	rs	us
2	sp	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	del

Casting between char and Numeric Types

```
int i = 'a'; // Same as int i = (int)'a';

char c = 97; // Same as char c = (char)97;
```

The `boolean` Type and Operators

Often in a program you need to compare two values, such as whether i is greater than j. Java provides six comparison operators (also known as relational operators) in Table 2.5 that can be used to compare two values. The result of the comparison is a Boolean value: true or false.

```
boolean b = (1 > 2);
```

Comparison Operators

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

Boolean Operators

Operator	*Name*
!	not
&&	and
||	or
^	exclusive or

Truth Table for Operator !

p	!p	Example
`true`	`false`	!(1 > 2) is true, because (1 > 2) is false.
`false`	`true`	!(1 > 0) is false, because (1 > 0) is true.

Truth Table for Operator &&

p1	p2	p1 && p2
false	false	false
false	true	false
true	false	false
true	true	true

Example

(3 > 2) && (5 >= 5) is true, because (3 > 2) and (5 >= 5) are both true.

(3 > 2) && (5 > 5) is false, because (5 > 5) is false.

Truth Table for Operator ||

p1	p2	p1 || p2
false	false	false
false	true	true
true	false	true
true	true	true

Example

(2 > 3) || (5 > 5) is false, because (2 > 3) and (5 > 5) are both false.

(3 > 2) || (5 > 5) is true, because (3 > 2) is true.

Truth Table for Operator ^

p1	p2	p1 ^ p2
false	false	false
false	true	true
true	false	true
true	true	false

Example

(2 > 3) ^ (5 > 1) is true, because (2 > 3) is false and (5 > 1) is true.

(3 > 2) ^ (5 > 1) is false, because both (3 > 2) and (5 > 1) are true.

Examples

```
System.out.println("Is " + num + " divisible by 2 and 3? " +
  ((num % 2 == 0) && (num % 3 == 0)));

System.out.println("Is " + num + " divisible by 2 or 3? " +
  ((num % 2 == 0) || (num % 3 == 0)));

 System.out.println("Is " + num +
   " divisible by 2 or 3, but not both? " +
   ((num % 2 == 0) ^ (num % 3 == 0)));
```

Leap Year?

A year is a leap year if it is divisible by 4 but not by 100 or if it is divisible by 400. The source code of the program is given below.

```
boolean isLeapYear =
    ((year % 4 == 0) && (year % 100 != 0)) ||
    (year % 400 == 0);
```

The & and | Operators

&&: conditional AND operator

&: unconditional AND operator

||: conditional OR operator

|: unconditional OR operator

exp1 && exp2

(1 < x) && (x < 100)

(1 < x) & (x < 100)

The & and | Operators

```
If x is 1, what is x after this
  expression?
(x > 1) & (x++ < 10)

If x is 1, what is x after this
  expression?
(1 > x) && ( 1 > x++)

How about (1 == x) | (10 > x++)?
(1 == x) || (10 > x++)?
```

Operator Precedence

How to evaluate 3 + 4 * 4 > 5 * (4 + 3) – 1?

Operator Precedence

- var++, var--
- +, - (Unary plus and minus), ++var,--var
- (type) Casting
- ! (Not)
- *, /, % (Multiplication, division, and remainder)
- +, - (Binary addition and subtraction)
- <, <=, >, >= (Comparison)
- ==, !=; (Equality)
- & (Unconditional AND)
- ^ (Exclusive OR)
- | (Unconditional OR)
- && (Conditional AND) Short-circuit AND
- || (Conditional OR) Short-circuit OR
- =, +=, -=, *=, /=, %= (Assignment operator)

Operator Precedence and Associativity

The expression in the parentheses is evaluated first. (Parentheses can be nested, in which case the expression in the inner parentheses is executed first.) When evaluating an expression without parentheses, the operators are applied according to the precedence rule and the associativity rule.

If operators with the same precedence are next to each other, their associativity determines the order of evaluation. All binary operators except assignment operators are left-associative.

Operator Associativity

When two operators with the same precedence are evaluated, the *associativity* of the operators determines the order of evaluation. All binary operators except assignment operators are *left-associative*.

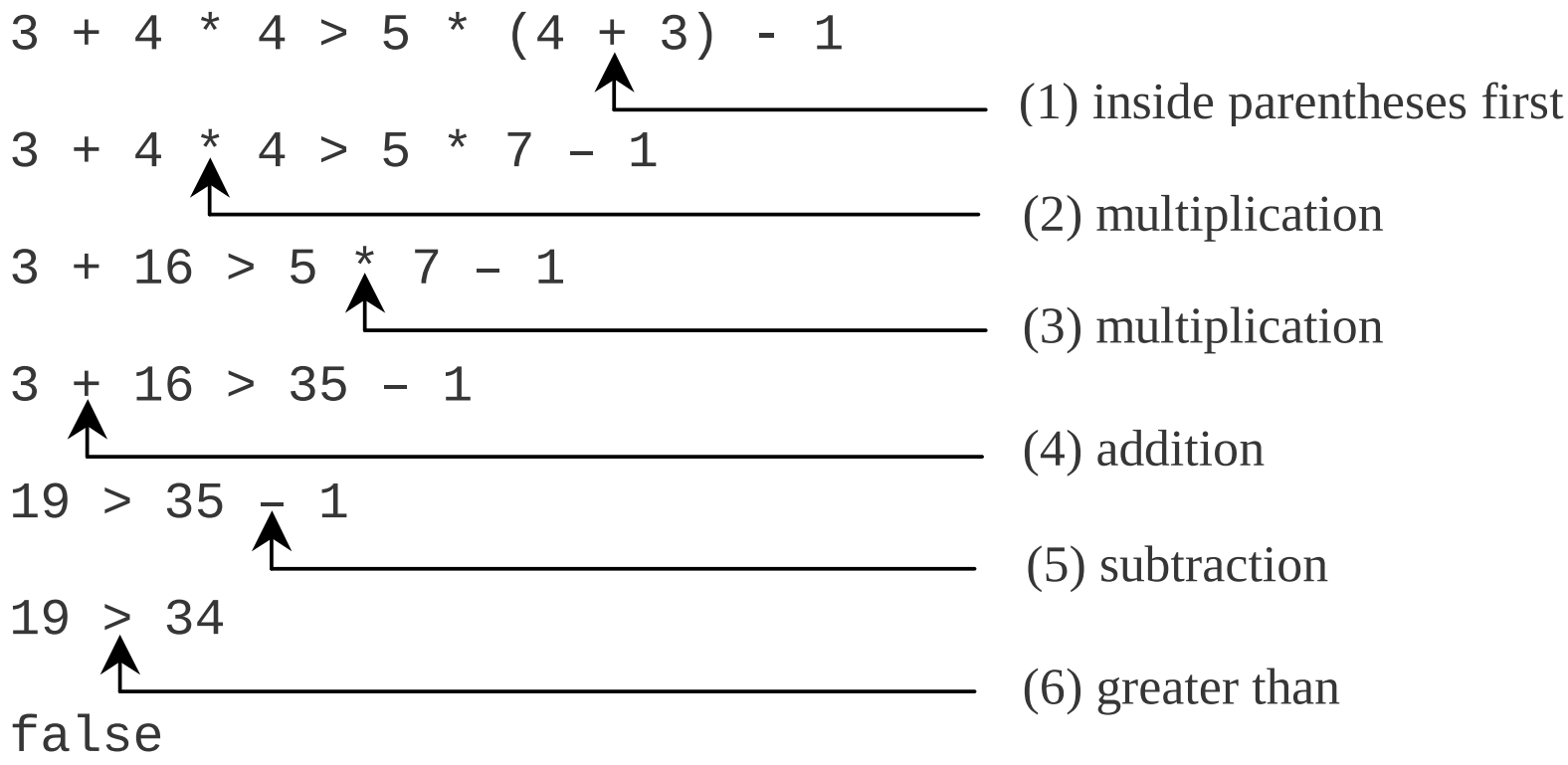
a – b + c – d is equivalent to ((a – b) + c) – d

Assignment operators are *right-associative*. Therefore, the expression

a = b += c = 5 is equivalent to a = (b += (c = 5))

Example

Applying the operator precedence and associativity rule, the expression 3 + 4 * 4 > 5 * (4 + 3) - 1 is evaluated as follows:



Operand Evaluation Order

The precedence and associativity rules specify the order of the operators, but do not specify the order in which the operands of a binary operator are evaluated. Operands are evaluated from left to right in Java.

The left-hand operand of a binary operator is evaluated before any part of the right-hand operand is evaluated.

Operand Evaluation Order, cont.

If no operands have *side effects* that change the value of a variable, the order of operand evaluation is irrelevant. Interesting cases arise when operands do have a side effect. For example, *x* becomes 1 in the following code, because a is evaluated to 0 before ++*a* is evaluated to 1.

```
int a = 0;
int x = a + (++a);
```

But *x* becomes 2 in the following code, because ++*a* is evaluated to 1, then *a* is evaluated to 1.

```
int a = 0;
int x = ++a + a;
```

Rule of Evaluating an Expression

· Rule 1: Evaluate whatever subexpressions you can possibly evaluate from left to right.

·

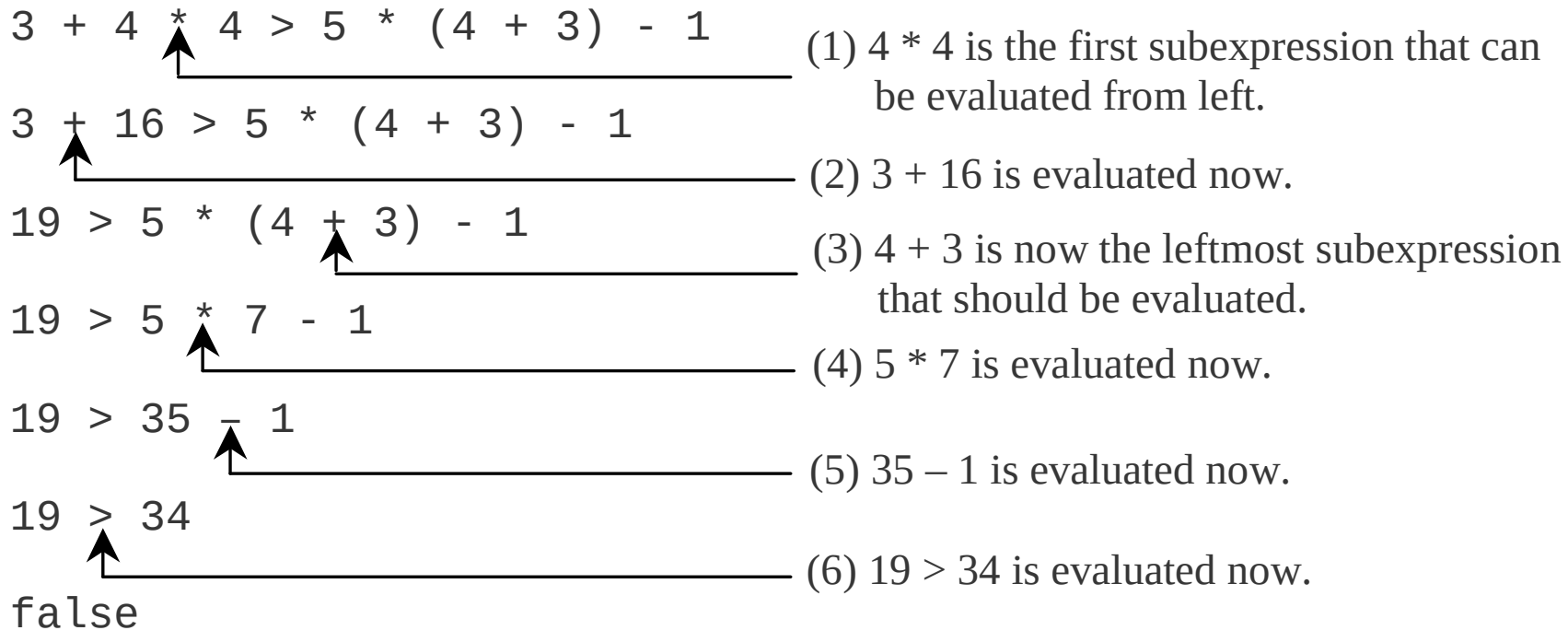
Rule 2: The operators are applied according to their precedence, as shown in Table 2.11.

·

Rule 3: The associativity rule applies for two operators next to each other with the same precedence.

Rule of Evaluating an Expression

- Applying the rule, the expression 3 + 4 * 4 > 5 * (4 + 3) - 1 is evaluated as follows:



The String Type

The char type only represents one character. To represent a string of characters, use the data type called String. For example,

String message = "Welcome to Java";

String is actually a predefined class in the Java library just like the System class and JOptionPane class. The String type is not a primitive type. It is known as a *reference type*. Any Java class can be used as a reference type for a variable. Reference data types will be thoroughly discussed in Chapter 6, “Classes and Objects.” For the time being, you just need to know how to declare a String variable, how to assign a string to the variable, and how to concatenate strings.

String Concatenation

```
// Three strings are concatenated
String message = "Welcome " + "to " + "Java";

// String Chapter is concatenated with number 2
String s = "Chapter" + 2; // s becomes Chapter2

// String Supplement is concatenated with character B
String s1 = "Supplement" + 'B'; // s becomes
SupplementB
```

Obtaining Input

This book provides three ways of obtaining input.

1. Using JOptionPane input dialogs (§2.14)
2. Using the JDK 1.5 Scanner class (Supplement T)
3. Using the MyInput class (§2.16)

Converting Strings to Integers

The input returned from the input dialog box is a string. If you enter a numeric value such as 123, it returns “123”. To obtain the input as a number, you have to convert a string into a number.

To convert a string into an int value, you can use the static parseInt method in the Integer class as follows:

int intValue = Integer.parseInt(intString);

where intString is a numeric string such as “123”.

Converting Strings to Doubles

To convert a string into a double value, you can use the static parseDouble method in the Double class as follows:

double doubleValue =Double.parseDouble(doubleString);

where doubleString is a numeric string such as “123.45”.

Example 2.3
Computing Loan Payments

This program lets the user enter the interest rate, number of years, and loan amount and computes monthly payment and total payment.

$$\frac{loanAmount \times monthlyInterestRate}{1 - \frac{1}{(1 + monthlyInterestRate)^{numOfYears \times 12}}}$$

Example 2.4 Monetary Units

This program lets the user enter the amount in decimal representing dollars and cents and output a report listing the monetary equivalent in single dollars, quarters, dimes, nickels, and pennies. Your program should report maximum number of dollars, then the maximum number of quarters, and so on, in this order.

Example 2.5
Displaying Current Time

Write a program that displays current time in GMT in the format hour:minute:second such as 1:45:19.

The currentTimeMillis method in the System class returns the current time in milliseconds since the midnight, January 1, 1970 GMT. (1970 was the year when the Unix operating system was formally introduced.) You can use this method to obtain the current time, and then compute the current second, minute, and hour as follows.

Optional Supplement T

Getting Input Using Scanner

1. Create a Scanner object

```
Scanner scanner = new Scanner(System.in);
```

2. Use the methods next(), nextByte(), nextShort(), nextInt(), nextLong(), nextFloat(), nextDouble(), or nextBoolean() to obtain to a string, byte, short, int, long, float, double, or boolean value. For example,

```
System.out.print("Enter a double value: ");
Scanner scanner = new Scanner(System.in);
double d = scanner.nextDouble();
```

TestScanner Run

JDK 1.5 Feature

Formatting Output

Use the new JDK 1.5 printf statement.

```
System.out.printf(format, item);
```

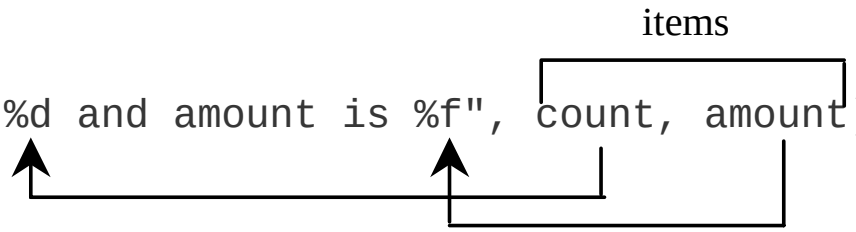
Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

JDK 1.5 Feature

Frequently-Used Specifiers

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"

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```




```
display            count is 5 and amount is 45.560000
```

Programming Style and Documentation

- Appropriate Comments
- Naming Conventions
- Proper Indentation and Spacing Lines
- Block Styles

Appropriate Comments

Include a summary at the beginning of the program to explain what the program does, its key features, its supporting data structures, and any unique techniques it uses.

Include your name, class section, instructor, date, and a brief description at the beginning of the program.

Naming Conventions

- Choose meaningful and descriptive names.
- Variables and method names:
  - Use lowercase. If the name consists of several words, concatenate all in one, use lowercase for the first word, and capitalize the first letter of each subsequent word in the name. For example, the variables `radius` and `area`, and the method `computeArea`.

Naming Conventions, cont.

- Class names:
  - Capitalize the first letter of each word in the name. For example, the class name `ComputeArea`.

- Constants:
  - Capitalize all letters in constants, and use underscores to connect words. For example, the constant `PI` and MAX_VALUE

Proper Indentation and Spacing

- Indentation
  - Indent two spaces.

- Spacing
  - Use blank line to separate segments of the code.

Block Styles

Use end-of-line style for braces.

Next-line style

```
public class Test
{
  public static void main(String[] args)
  {
    System.out.println("Block Styles");
  }
}
```

End-of-line style

```
public class Test {
  public static void main(String[] args) {
    System.out.println("Block Styles");
  }
}
```

Programming Errors

- Syntax Errors
  - Detected by the compiler
- Runtime Errors
  - Causes the program to abort
- Logic Errors
  - Produces incorrect result

Syntax Errors

```
public class ShowSyntaxErrors {
  public static void main(String[] args) {
    i = 30;
    System.out.println(i + 4);
  }
}
```

Runtime Errors

```
public class ShowRuntimeErrors {
  public static void main(String[] args) {
    int i = 1 / 0;
  }
}
```

Logic Errors

```
public class ShowLogicErrors {
  // Determine if a number is between 1 and 100 inclusively
  public static void main(String[] args) {
    // Prompt the user to enter a number
    String input = JOptionPane.showInputDialog(null,
      "Please enter an integer:",
      "ShowLogicErrors", JOptionPane.QUESTION_MESSAGE);
    int number = Integer.parseInt(input);

    // Display the result
    System.out.println("The number is between 1 and 100, " +
      "inclusively? " + ((1 < number) && (number < 100)));

    System.exit(0);
  }
}
```

Debugging

- Logic errors are called *bugs*.
- The process of finding and correcting errors is called debugging.
- Debugging could be done through a combination of methods to narrow down to the part of the program where the bug is located.
  - Hand-tracing
  - Insert print statements in order to show the values of the variables or the execution flow of the program.
  - Use debugger

Debugger

Debugger is a program that facilitates debugging. You can use a debugger to

- Execute a single statement at a time.
- Trace into or stepping over a method.
- Set breakpoints.
- Display variables.
- Display call stack.
- Modify variables.