Elementary Programming
Imperative Programming

To write Java programs to perform simple calculations
To obtain input from the console using the Scanner class
To use identifiers to name variables, constants, methods, and classes
To use variables to store data
To program with assignment statements and assignment expressions
To use constants to store permanent data
To declare Java primitive data types: byte, short, int, long, float, double, and char
To use Java operators to write numeric expressions
To display current time
To use short hand operators
To cast value of one type to another type
To represent characters using the char type
To represent a string using the String type
To become familiar with Java documentation, programming style, and naming conventions
To input/output using the JOptionPane input dialog boxes

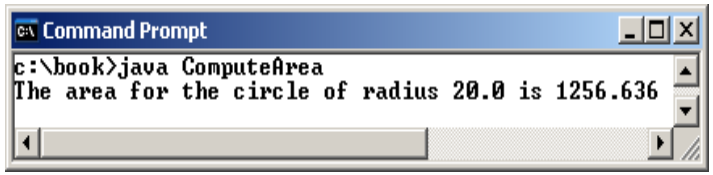
Computing the Area of a Circle

```
public class ComputeArea {
  /** Main method */
  public static void main(String[] args) {
    double radius;
    double area;

    // Assign a radius
    radius = 20;

    // Compute area
    area = radius * radius * 3.14159;

    // Display results
    System.out.println("The area for the circle of radius " +
      radius + " is " + area);
  }
}
```

allocate memory for radius

radius	no value
area	no value

radius	20
area	1256.636



Reading Input from the Console

1. Create a Scanner object

```
Scanner input = 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 input = new Scanner(System.in);
double d = input.nextDouble();
```

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.
  - 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;`

Constants

```
final datatype CONSTANTNAME = VALUE;

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

Numerical Data Types

Name	Range	Storage Size
byte	-2^7 (-128) to 2^7-1 (127)	8-bit signed
short	-2^{15} (-32768) to $2^{15}-1$ (32767)	16-bit signed
int	-2^{31} (-2147483648) to $2^{31}-1$ (2147483647)	32-bit signed
long	-2^{63} to $2^{63}-1$ (i.e., -9223372036854775808 to 9223372036854775807)	64-bit signed
float	Negative range: -3.4028235E+38 to -1.4E-45 Positive range: 1.4E-45 to 3.4028235E+38	32-bit IEEE 754
double	Negative range: -1.7976931348623157E+308 to -4.9E-324 Positive range: 4.9E-324 to 1.7976931348623157E+308	64-bit IEEE 754

Arithmetic/Numeric Operators

Name	Meaning	Example	Result
+	Addition	34 + 1	35
-	Subtraction	34.0 – 0.1	33.9
*	Multiplication	300 * 30	9000
/	Division	1.0 / 2.0	0.5
%	Remainder	20 % 3	2

Integer Division

+, -, *, /, and %

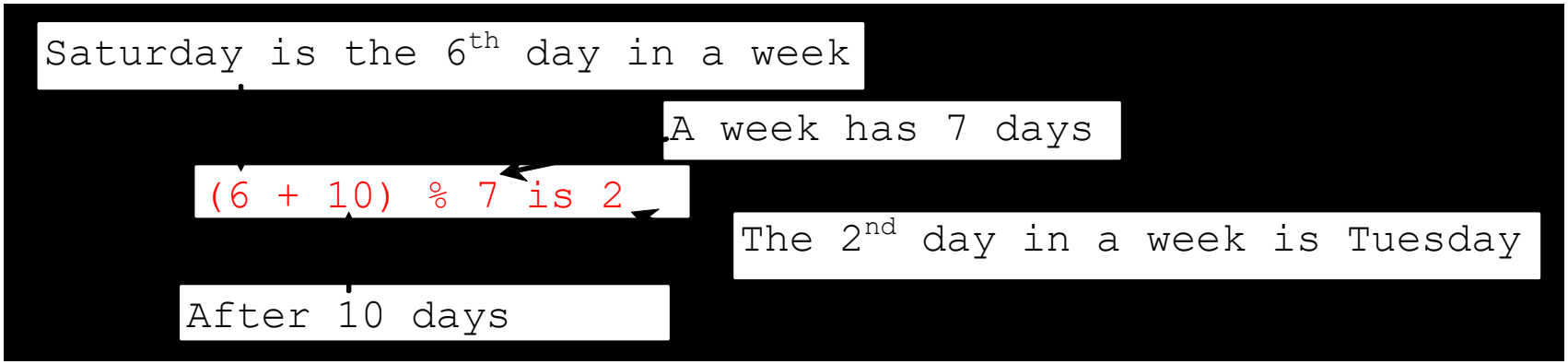
5 / 2 yields an integer 2.

5.0 / 2 yields a double value 2.5

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

Remainder Operator

Remainder is very useful in programming. For example, an even number % 2 is always 0 and an odd number % 2 is always 1. So you can use this property to determine whether a number is even or odd. Suppose today is Saturday and you and your friends are going to meet in 10 days. What day is in 10 days? You can find that day is Tuesday using the following expression:



Number Literals

A *literal* is a fixed 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.

How to Evaluate an Expression

Though Java has its own way to evaluate an expression behind the scene, the result of a Java expression and its corresponding arithmetic expression are the same. Therefore, you can safely apply the arithmetic rule for evaluating a Java expression.

```
3 + 4 * 4 + 5 * (4 + 3) - 1
                        (1) inside parentheses first
3 + 4 * 4 + 5 * 7 - 1
                        (2) multiplication
3 + 16 + 5 * 7 - 1
                        (3) multiplication
3 + 16 + 35 - 1
                        (4) addition
19 + 35 - 1
                        (5) addition
  54 - 1
                        (6) subtraction
  53
```

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))

Operator Precedence & Associativity Table

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type) * & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= >>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Precedence and Associativity Table

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

Example

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

```
3 + 4 * 4 > 5 * (4 + 3) - 1
```

(1) inside parentheses first

```
3 + 4 * 4 > 5 * 7 - 1
```

(2) multiplication

```
3 + 16 > 5 * 7 - 1
```

(3) multiplication

```
3 + 16 > 35 - 1
```

(4) addition

```
19 > 35 - 1
```

(5) subtraction

```
19 > 34
```

(6) greater than

```
false
```

Problem: Converting Temperatures

Write a program that converts a Fahrenheit degree to Celsius using the formula:

$$celsius = (\tfrac{5}{9})(fahrenheit - 32)$$

Problem: Displaying Current Time

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

Current Time Calculation

```
public class ShowCurrentTime {
  public static void main(String[] args) {
      // Obtain the total milliseconds since midnight, Jan 1, 1970
      long totalMilliseconds = System.currentTimeMillis();
      long totalSeconds = totalMilliseconds / 1000;
      long currentSecond = (int)(totalSeconds % 60); // Obtain the total minutes
      long totalMinutes = totalSeconds / 60; // Compute the current minute in the hour
      long currentMinute = (int)(totalMinutes % 60); // Obtain the total hours
      long totalHours = totalMinutes / 60; // Compute the current hour
      long currentHour = (int)(totalHours % 24); // Display results
      System.out.println("Current time is " + currentHour + ":" + currentMinute + ":" +
      currentSecond + " GMT");
  }
}
```

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	The expression (++var) increments var by 1 and evaluates to the *new* value in var *after* the increment.
var++	postincrement	The expression (var++) evaluates to the *original* value in var and increments var by 1.
--var	predecrement	The expression (--var) decrements var by 1 and evaluates to the *new* value in var *after* the decrement.
var--	postdecrement	The expression (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;
```

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;

range increases

byte, short, int, long, float, double

Character Data Type

Four hexadecimal digits.

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

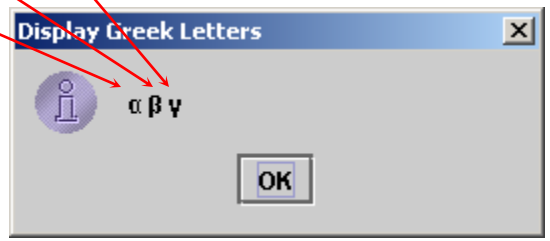
NOTE: The increment and decrement operators can also be used on <u>char</u> variables to get the next or preceding Unicode character. For example, the following statements display character <u>b</u>.

```
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`.

Unicode \u03b1 \u03b2 \u03b3 for three Greek letters



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`

Casting between char and Numeric Types

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

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

Problem: 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 cents. Your program should report maximum number of dollars, then the maximum number of quarters, and so on, in this order.

Trace ComputeChange

Suppose amount is 11.56

```
int remainingAmount = (int)(amount * 100);

// Find the number of one dollars
int numberOfOneDollars = remainingAmount / 100;
remainingAmount = remainingAmount % 100;

// Find the number of quarters in the remaining amount
int numberOfQuarters = remainingAmount / 25;
remainingAmount = remainingAmount % 25;

// Find the number of dimes in the remaining amount
int numberOfDimes = remainingAmount / 10;
remainingAmount = remainingAmount % 10;

// Find the number of nickels in the remaining amount
int numberOfNickels = remainingAmount / 5;
remainingAmount = remainingAmount % 5;

// Find the number of pennies in the remaining amount
int numberOfPennies = remainingAmount;
```

remainingAmount	1156
numberOfOneDollars	11
remainingAmount	56
numberOfQuarters	2
remainingAmount	6
numberOfDimes	0
remainingAmount	6
numberOfNickels	1
remainingAmount	1
numberOfPennies	1

Bitwise Operators

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

The String Type

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

<u>String message = "Welcome to Java";</u>

<u>String</u> is actually a predefined class in the Java library just like the <u>System</u> class and <u>JOptionPane</u> class. The <u>String</u> 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 7, “Objects and Classes.” For the time being, you just need to know how to declare a <u>String</u> 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'; // s1 becomes SupplementB
```

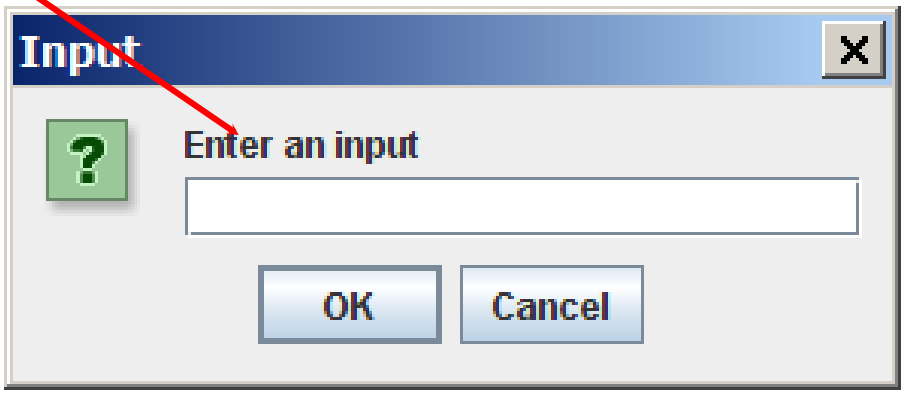
JOptionPane Input

This book provides two ways of obtaining input.

1. Using the Scanner class (console input)
2. Using JOptionPane input dialogs

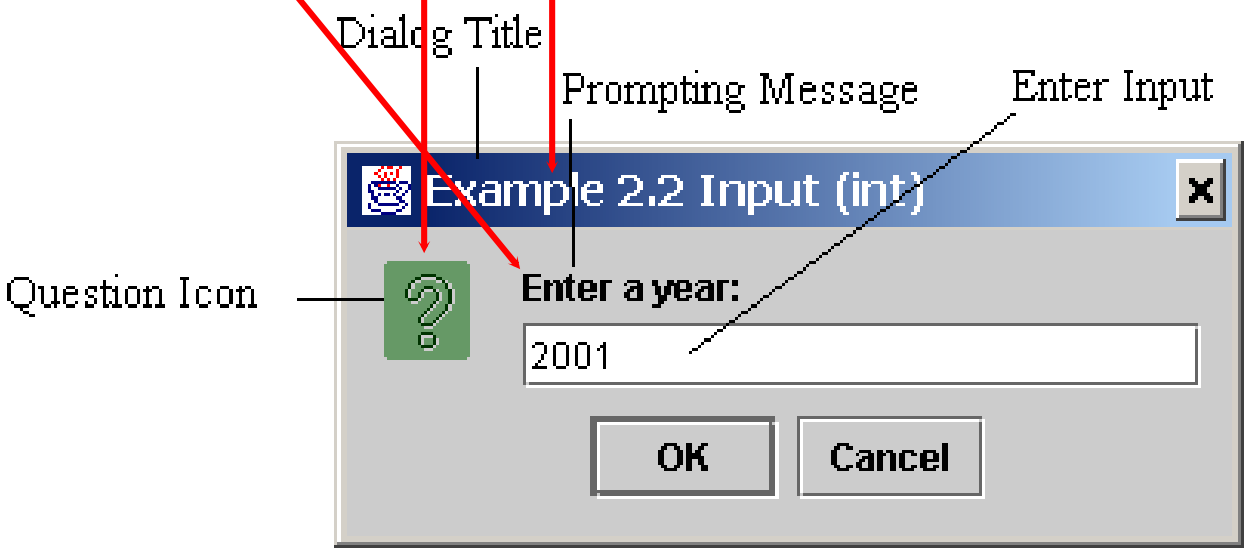
Getting Input from Input Dialog Boxes

```
String input =  JOptionPane.showInputDialog(
  "Enter an input");
```



Getting Input from Input Dialog Boxes

```
String string = JOptionPane.showInputDialog(
  null, “Prompting Message”,  “Dialog Title”,
  JOptionPane.QUESTION_MESSAGE);
```



Two Ways to Invoke the Method

There are several ways to use the showInputDialog method. For the time being, you only need to know two ways to invoke it.

One is to use a statement as shown in the example:

```
String string = JOptionPane.showInputDialog(null, msg,
  title, JOptionPane.QUESTION_MESSAGE);
```

where msg is a string for the prompting message, and title is a string for the title of the input dialog box.

The other is to use a statement like this:

```
JOptionPane.showInputDialog(msg);
```

where x is a string for the prompting message.

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”.

The showMessageDialog Method

```
JOptionPane.showMessageDialog(null,
  "Welcome to Java!",
  "Display Message",
  JOptionPane.INFORMATION_MESSAGE);
```

Two Ways to Invoke the Method

There are several ways to use the showMessageDialog method. For the time being, all you need to know are two ways to invoke it.

One is to use a statement as shown in the example:

```
JOptionPane.showMessageDialog(null, msg,
  title, JOptionPane.INFORMATION_MESSAGE);
```

The other is to use a statement like this:

```
JOptionPane.showMessageDialog(null, msg);
```

where x is a string for the text to be displayed.

(GUI) Confirmation Dialogs

```
int option = JOptionPane.showConfirmDialog
    (null, "Continue");
```