Basic Computation

Chapter 2
Objectives

• Describe the Java data types used for simple data

• Write Java statements to declare variables, define named constants

• Write assignment statements, expressions containing variables and constants
Objectives

• Write Java statements that accomplish keyboard input, screen output
• Adhere to stylistic guidelines and conventions
Variables

- Variables store data such as numbers and letters.
  - Think of them as places to store data.
  - They are implemented as memory locations.
- The data stored by a variable is called its value.
  - The value is stored in the memory location.
- Its value can be changed.
Variables

• View sample program listing 2.1

  - Class EggBasket

If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Variables and Values

• Variables
  
  ```java
  numberOfBaskets
eggsPerBasket
totalEggs
  ```

• Assigning values
  
  ```java
  eggsPerBasket = 6;
eggsPerBasket = eggsPerBasket - 2;
  ```
Naming and Declaring Variables

- Choose names that are helpful such as `count` or `speed`, but not `c` or `s`.
- When you *declare* a variable, you provide its type and name.
  ```java
  int numberOfBaskets, eggsPerBasket;
  ```
- A variable's *type* determines what kinds of values it can hold (`int`, `double`, `char`, etc.).
- A variable must be declared before it is used.
Syntax and Examples

• Syntax

    type variable_1, variable_2, ...;
    (variable_1 is a generic variable called a syntactic variable)

• Examples

    int styleChoice, numberOfChecks;
    double balance, interestRate;
    char jointOrIndividual;
Data Types

• A class type is used for a class of objects and has both data and methods.
  - "Java is fun" is a value of class type String

• A primitive type is used for simple, nondecomposable values such as an individual number or individual character.
  - int, double, and char are primitive types.
### Primitive Types

- **Figure 2.1 Primitive Types**

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Kind of Value</th>
<th>Memory Used</th>
<th>Range of Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>Integer</td>
<td>1 byte</td>
<td>(-128 ) to (127)</td>
</tr>
<tr>
<td>short</td>
<td>Integer</td>
<td>2 bytes</td>
<td>(-32,768 ) to (32,767)</td>
</tr>
<tr>
<td>int</td>
<td>Integer</td>
<td>4 bytes</td>
<td>(-2,147,483,648 ) to (2,147,483,647)</td>
</tr>
<tr>
<td>long</td>
<td>Integer</td>
<td>8 bytes</td>
<td>(-9,223,372,036,854,75,808 ) to (9,223,372,036,854,775,807)</td>
</tr>
<tr>
<td>float</td>
<td>Floating-point</td>
<td>4 bytes</td>
<td>(\pm 3.40282347 \times 10^{38} ) to (\pm 1.40239846 \times 10^{-45})</td>
</tr>
<tr>
<td>double</td>
<td>Floating-point</td>
<td>8 bytes</td>
<td>(\pm 1.79769313486231570 \times 10^{308} ) to (\pm 4.94065645841246544 \times 10^{-324})</td>
</tr>
<tr>
<td>char</td>
<td>Single character</td>
<td>2 bytes</td>
<td>All Unicode values from 0 to 65,535</td>
</tr>
<tr>
<td></td>
<td>(Unicode)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td></td>
<td>1 bit</td>
<td>True or false</td>
</tr>
</tbody>
</table>
Examples of Primitive Values

• Integer types
  
  0  -1  365  12000

• Floating-point types
  
  0.99  -22.8  3.14159  5.0

• Character type
  
  'a'  'A'  '#'  '

• Boolean type
  
  true  false
Java Identifiers

• An *identifier* is a name, such as the name of a variable.

• Identifiers may contain only
  ▪ Letters
  ▪ Digits (0 through 9)
  ▪ The underscore character (_)
  ▪ And the dollar sign symbol ($) which has a special meaning

• The first character *cannot* be a digit.
Java Identifiers

- Identifiers may not contain any spaces, dots (.), asterisks (*), or other characters:
  
  7-11 netscape.com util.* (not allowed)

- Identifiers can be arbitrarily long.

- Since Java is case sensitive, stuff, Stuff, and STUFF are different identifiers.
Keywords or Reserved Words

• Words such as `if` are called `keywords` or `reserved words` and have special, predefined meanings.
  ▪ Cannot be used as identifiers.
  ▪ See Appendix 1 for a complete list of Java keywords.

• Example keywords: `int`, `public`, `class`
Naming Conventions

• Class types begin with an uppercase letter (e.g. `String`).

• Primitive types begin with a lowercase letter (e.g. `int`).

• Variables of both class and primitive types begin with a lowercase letters (e.g. `myName`, `myBalance`).

• Multiword names are "punctuated" using uppercase letters.
Where to Declare Variables

• Declare a variable
  ▪ Just before it is used or
  ▪ At the beginning of the section of your program that is enclosed in `{ }.

```java
public static void main(String[] args) {
    /* declare variables here */
    . . .
}
```
Assignment Statements

• An assignment statement is used to assign a value to a variable.

    \texttt{answer = 42;}

• The "equal sign" is called the \textit{assignment operator}.

• We say, "The variable named \texttt{answer} is assigned a value of 42," or more simply, "\texttt{answer} is assigned 42."
Assignment Statements

• Syntax

\[\text{variable} = \text{expression}\]

where \text{expression} can be another variable, a \emph{literal} or \emph{constant} (such as a number), or something more complicated which combines variables and literals using \emph{operators} (such as + and -)
Assignment Examples

```java
amount = 3.99;
firstInitial = 'W';
score = numberOfCards + handicap;
eggsPerBasket = eggsPerBasket - 2;
```
Initializing Variables

• A variable that has been declared, but not yet given a value is said to be \textit{uninitialized}.
• Uninitialized class variables have the value \texttt{null}.
• Uninitialized primitive variables may have a default value.
• It's good practice not to rely on a default value.
Initializing Variables

• To protect against an uninitialized variable (and to keep the compiler happy), assign a value at the time the variable is declared.

• Examples:

```java
int count = 0;
char grade = 'A';
```
Constants

• Literal expressions such as $2, 3.7, \text{or} \ 'y' $ are called constants.
• Integer constants can be preceded by a $+$ or $-$ sign, but cannot contain commas.
• Floating-point constants can be written
  ▪ With digits after a decimal point or
  ▪ Using e notation.
e Notation

- e notation is also called **scientific notation** or **floating-point notation**.
- Examples
  - $865000000.0$ can be written as $8.65e8$
  - $0.000483$ can be written as $4.83e-4$
- The number in front of the e does not need to contain a decimal point.
Imprecision in Floating-Point Numbers

• Floating-point numbers often are only approximations since they are stored with a finite number of bits.
• Hence \( \frac{1.0}{3.0} \) is slightly less than \( \frac{1}{3} \).
• \( \frac{1.0}{3.0} + \frac{1.0}{3.0} + \frac{1.0}{3.0} \) is less than 1.
Named Constants

- Java provides mechanism to ... 
  - Define a variable 
  - Initialize it 
  - Fix the value so it cannot be changed 

```java
public static final Type Variable = Constant;
```

- Example 

```java
public static final double PI = 3.14159;
```
Assignment Compatibilities

- Java is said to be *strongly typed*.
  - You can't, for example, assign a floating point value to a variable declared to store an integer.

- Sometimes conversions between numbers are possible.

```java
doubleValue = 7;
```

is possible even if `doubleValue` is of type `double`.
Assignment Compatibilities

• A value of one type can be assigned to a variable of any type further to the right
  byte --> short --> int --> long
  --> float --> double
  ▪ But not to a variable of any type further to the left.

• You can assign a value of type char to a variable of type int.
Type Casting

• A *type cast* temporarily changes the value of a variable from the declared type to some other type.

• For example,

```
double distance;
distance = 9.0;
int points;
points = (int)distance;
```

• Illegal without `(int)`
Type Casting

• The value of \((\text{int})\text{distance}\) is 9,
• The value of \text{distance}, both before and after the cast, is 9.0.
• Any nonzero value to the right of the decimal point is \textit{truncated} rather than \textit{rounded}.
Arithmetic Operators

• Arithmetic expressions can be formed using the +, −, *, and / operators together with variables or numbers referred to as operands.
  ▪ When both operands are of the same type, the result is of that type.
  ▪ When one of the operands is a floating-point type and the other is an integer, the result is a floating point type.
Arithmetic Operations

• Example

If `hoursWorked` is an `int` to which the value `40` has been assigned, and `payRate` is a `double` to which `8.25` has been assigned

`hoursWorked * payRate`

is a `double` with a value of `500.0`. 
Arithmetic Operations

• Expressions with two or more operators can be viewed as a series of steps, each involving only two operands.
  - The result of one step produces one of the operands to be used in the next step.

• example

  \texttt{balance + (balance * rate)}
Arithmetic Operations

• If at least one of the operands is a floating-point type and the rest are integers, the result will be a floating point type.

• The result is the rightmost type from the following list that occurs in the expression.
  byte --> short --> int --> long --> float --> double
The Division Operator

• The division operator (/) behaves as expected if one of the operands is a floating-point type.
• When both operands are integer types, the result is truncated, not rounded.
  ▪ Hence, 99/100 has a value of 0.
**The mod Operator**

- The mod (\%) operator is used with operators of integer type to obtain the remainder after integer division.
- 14 divided by 4 is 3 *with a remainder of 2.*
  - Hence, $14 \% 4$ is equal to 2.
- The mod operator has many uses, including
  - determining if an integer is odd or even
  - determining if one integer is evenly divisible by another integer.
Parentheses and Precedence

• Parentheses can communicate the order in which arithmetic operations are performed

• examples:

  $(\text{cost} + \text{tax}) \times \text{discount}$
  $(\text{cost} + (\text{tax} \times \text{discount}))$

• Without parentheses, an expression is evaluated according to the rules of precedence.
Precedence Rules

- Figure 2.2 Precedence Rules

**Highest Precedence**

First: the unary operators +, -, !, ++, and --
Second: the binary arithmetic operators *, /, and %
Third: the binary arithmetic operators + and −

**Lowest Precedence**
Precedence Rules

• The *binary* arithmetic operators *, /, and %, have *lower precedence* than the *unary* operators +, −, ++, --, and !, but have *higher precedence* than the binary arithmetic operators + and −.

• When binary operators have equal precedence, the operator on the left acts before the operator(s) on the right.
Precedence Rules

• When unary operators have equal precedence, the operator on the right acts before the operation(s) on the left.

• Even when parentheses are not needed, they can be used to make the code clearer.

  balance + (interestRate * balance)

• Spaces also make code clearer

  balance + interestRate*balance

but spaces do not dictate precedence.
Sample Expressions

- Figure 2.3 Some Arithmetic Expressions in Java

<table>
<thead>
<tr>
<th>Ordinary Math</th>
<th>Java (Preferred Form)</th>
<th>Java (Parenthesized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( rate^2 + delta )</td>
<td>( rate \times rate + delta )</td>
<td>((rate \times rate) + delta)</td>
</tr>
<tr>
<td>( 2(salary + bonus) )</td>
<td>( 2 \times (salary + bonus) )</td>
<td>( 2 \times (salary + bonus) )</td>
</tr>
<tr>
<td>( \frac{1}{time + 3mass} )</td>
<td>( 1 \div (time + 3 \times mass) )</td>
<td>( 1 \div (time + (3 \times mass)) )</td>
</tr>
<tr>
<td>( \frac{a - 7}{t + 9v} )</td>
<td>( (a - 7) \div (t + 9 \times v) )</td>
<td>( (a - 7) \div (t + (9 \times v)) )</td>
</tr>
</tbody>
</table>
Specialized Assignment Operators

• Assignment operators can be combined with arithmetic operators (including -, *, /, and %, discussed later).

\[
\text{amount} = \text{amount} + 5;
\]

can be written as

\[
\text{amount} += 5;
\]

yielding the same results.
Simple Input

• Sometimes the data needed for a computation are obtained from the user at run time.

• Keyboard input requires

import java.util.Scanner

at the beginning of the file.
Simple Input

• Data can be entered from the keyboard using
  ```java
  Scanner keyboard = new Scanner(System.in);
  ```
  followed, for example, by
  ```java
  eggsPerBasket = keyboard.nextInt();
  ```
  which reads one `int` value from the keyboard and assigns it to `eggsPerBasket`. 
Simple Input

• View sample program listing 2.2

class EggBasket2

Enter the number of eggs in each basket:
6
Enter the number of baskets:
10
If you have
6 eggs per basket and
10 baskets, then
the total number of eggs is 60
Now we take two eggs out of each basket.
You now have
4 eggs per basket and
10 baskets.
The new total number of eggs is 40
Simple Screen Output

System.out.println("The count is " + count);

• Outputs the string literal "the count is "
• Followed by the current value of the variable count.
Case Study: Vending Machine Change

• Requirements
  ▪ The user enters an amount between 1 cent and 99 cents.
  ▪ The program determines a combination of coins equal to that amount.
  ▪ For example, 55 cents can be two quarters and one nickel.
Case Study

• Sample dialog

Enter a whole number from 1 to 99. The machine will determine a combination of coins.

87

87 cents in coins:

3 quarters
1 dime
0 nickels
2 pennies
Case Study

• Variables needed

```java
int amount,
quarters,
dimes,
nickels,
pennies;
```
Case Study

• Algorithm - first version

1. Read the amount.
2. Find the maximum number of quarters in the amount.
3. Subtract the value of the quarters from the amount.
4. Repeat the last two steps for dimes, nickels, and pennies.
5. Print the original amount and the quantities of each coin.
Case Study, cont.

• The algorithm doesn't work properly
  ▪ Original amount is changed by the intermediate steps.
  ▪ Original value of `amount` is lost.

• Change the list of variables

```java
int amount, originalAmount, quarters, dimes, nickles, pennies;
```
  ▪ Update the algorithm.
Case Study

• Algorithm – second version

1. Read the amount.
2. Make a copy of the amount.
3. Find the maximum number of quarters in the amount.
4. Subtract the value of the quarters from the amount.
5. Repeat the last two steps for dimes, nickels, and pennies.
6. Print the original amount and the quantities of each coin.
Case Study

• View Java code that implements the algorithm written in pseudocode – listing 2.3

Enter a whole number from 1 to 99. I will find a combination of coins that equals that amount of change.

87

87 cents in coins can be given as:
3 quarters
1 dimes
0 nickels and
2 pennies
Case Study

• How do we determine the number of quarters (or dimes, nickels, or pennies) in an amount?

\[ 55 \div 25 = 2 \text{ and } 65 \div 25 = 2. \]
Case Study

• How do we determine the remaining amount?
• The remaining amount can be determined using the mod operator
  \[ 55 \% \ 25 = 5 \text{ and } 65 \% \ 25 = 15 \]
• Similarly for dimes and nickels.
• Pennies are simply \texttt{amount} \ % \ 5.