Basic Computation

Chapter 2
Edited by JJ Shepherd, James O’Reilly
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
• Define strings of characters, perform simple string processing
Objectives

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

• Variables and Expressions
• The Class **String**
• Keyboard and Screen I/O
• Documentation and Style
Variables and Expressions: Outline

• Variables
• Data Types
• Java Identifiers
• Assignment Statements
• Simple Input
• Simple Screen Output
• Constants
• Named Constants
Variables and Expressions: Outline

• Assignment Compatibilities
• Type Casting
• Arithmetic Operations
• Parentheses and Precedence Rules
• Specialized Assignment Operators
• Case Study: Vending Machine Change
• Increment and Decrement Operators
Variables

- **Variables** store data such as numbers and letters.
  - Think of them as places to store data.
  - They are implemented as memory locations.

```java
int numberOfCats = 1;
```
Variables

• The data stored by a variable is called its **value**.
  • The value is stored in the memory location.
• Its value can be changed.

```java
int numberOfCats = 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 name and type.

```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

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

• Examples

```java
    int styleChoice, numberOfChecks;
    double balance, interestRate;
    char jointOrIndividual;
```
THE BIG IDEA

• To declare variables you provide its TYPE and NAME

```java
int numberOfCats;
```

• Let’s look at these in more detail starting with...
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

• Four integer types (byte, short, int, and long)
  • int is most common
• Two floating-point types (float and double)
  • double is more common
• One character type (char)
• One boolean type (boolean)
Examples of Primitive Values

• Integer types
  0  -1  365  12000

• Floating-point types
  0.99  -22.8  3.14159  5.0

• Character type: single quotes
  'a'  'A'  '#'  '

• Boolean type
  true  false
## Primitive Types

### FIGURE 2.1 Primitive Type

<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>±3.40282347 \times 10^{+38} to ±1.40239846 \times 10^{-45}</td>
</tr>
<tr>
<td>double</td>
<td>Floating-point</td>
<td>8 bytes</td>
<td>±1.79769313486231570 \times 10^{+308} to ±4.94065645841246544 \times 10^{-324}</td>
</tr>
<tr>
<td>char</td>
<td>Single character (Unicode)</td>
<td>2 bytes</td>
<td>All Unicode values from 0 to 65,535</td>
</tr>
<tr>
<td>boolean</td>
<td></td>
<td>1 bit</td>
<td>True or false</td>
</tr>
</tbody>
</table>
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 oracle.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`
SYNTAX ERROR!
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.
Naming Conventions

• The method for lower casing the first letter in the name and then uppercasing the first letter in each word is sometimes called “Camel Casing”
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.
  \[
  \text{answer} = 42;
  \]

• The "equal sign" is called the assignment operator.

• \textit{It is not the same as mathematical equality.}

• The value on the left will only be equal at the end of the statement until you change the value later.

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

• Syntax

```java
variable = expression
```

where `expression` can be another variable, a literal or constant (such as a number), or something more complicated which combines variables and literals using `operators` (such as + and -)

The value on the left must be a variable

```java
2= a //BAD!
```

Will raise an error
Assignment Examples

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

• A variable that has been declared, but no yet given a value is said to be uninitialized.
• Uninitialized class variables have the value 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:

  int count = 0;
  char grade = 'A';
Initializing Variables

• syntax

  type variable_1 = expression_1, variable_2 = expression_2, ...;

  int powerLevel = 9001;
Assignment Evaluation

• The expression on the right-hand side of the assignment operator (\(=\)) is evaluated first.
• The result is used to set the value of the variable on the left-hand side of the assignment operator.

\[
\text{score} = \text{numberOfCards} + \text{handicap};
\]
\[
\text{eggsPerBasket} = \text{eggsPerBasket} - 2;
\]
INPUT
OUTPUT
Simple Input

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

```java
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 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.
MORE STUFF!
Constants

• Literal expressions such as 2, 3.7, 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.
Named Constants

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

public static final Type Variable = Constant;

• Example

public static final double PI = 3.14159;
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 \( 1.0/3.0 \) is slightly less than \( 1/3 \).
• \( 1.0/3.0 + 1.0/3.0 + 1.0/3.0 \) is less than 1.
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.
  • When you declare a variable you must give its type.

• Sometimes conversions between numbers are possible.

```java
doubleValue = 7;
```

is possible even if `doubleValue` is of type `double`, for example.
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,

```java
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 `330.0`.
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.
  • Regular order of operations (~PEMDAS)

• example
  \[ \text{balance} + (\text{balance} \times \text{rate}) \]
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, \(14 \mod 4\) 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.
  • Integer division(/) gives results without remainder and mod, next, gives remainder... together useful