Basic Computation

Chapter 2 Part 2

Edited by JJ Shepherd, James O'Reilly

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119

Parentheses and Precedence

- Parentheses can communicate the order in which arithmetic operations are performed
- examples:

```
(cost + tax) * discount
(cost + (tax * discount)
```

- Without parentheses, an expressions 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

Ordinary Math	Java (Preferred Form)	Java (Parenthesized)
$rate^2 + delta$	`rate * rate + delta`	`(rate * rate) + delta`
$2(salary + bonus)$	`2 * (salary + bonus)`	`2 * (salary + bonus)`
$\frac{1}{time + 3mass}$	`1 / (time + 3 * mass)`	`1 / (time + (3 * mass))`
$\frac{a-7}{t+9v}$	`(a - 7) / (t + 9 * v)`	`(a - 7) / (t + (9 * v))`

Specialized Assignment Operators

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

```
amount = amount + 5;
```

can be written as

```
amount += 5;
```

yielding the same results.

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.

Increment and Decrement Operators

- Used to increase (or decrease) the value of a variable by 1
- Easy to use, important to recognize
- The increment operator

  `count++` or `++count`
- The decrement operator

  `count--` or `--count`

Increment and Decrement Operators

- equivalent operations

```
count++;
++count;
count = count + 1;

count--;
--count;
count = count - 1;
```

Increment and Decrement Operators in Expressions

- after executing

```
int m = 4;
int result = 3 * (++m)
```

`result` has a value of `15` and `m` has a value of `5`

- after executing

```
int m = 4;
int result = 3 * (m++)
```

`result` has a value of `12` and `m` has a value of `5`

*JAVA: An Introduction to Problem Solving & Programming, 7*th Ed. By Walter Savitch
ISBN 0133862119

The Class `String`

- We've used constants of type `String` already.

```
"Enter a whole number from 1 to 99."
```

- A value of type `String` is a
  - Sequence of characters
  - Treated as a single item.

String Constants and Variables

- Declaring

```
String greeting;
```

```
greeting = "Hello!";
```

or

```
String greeting = "Hello!";
```

or

```
String greeting = new String("Hello!");
```

- Printing

```
System.out.println(greeting);
```

Concatenation of Strings

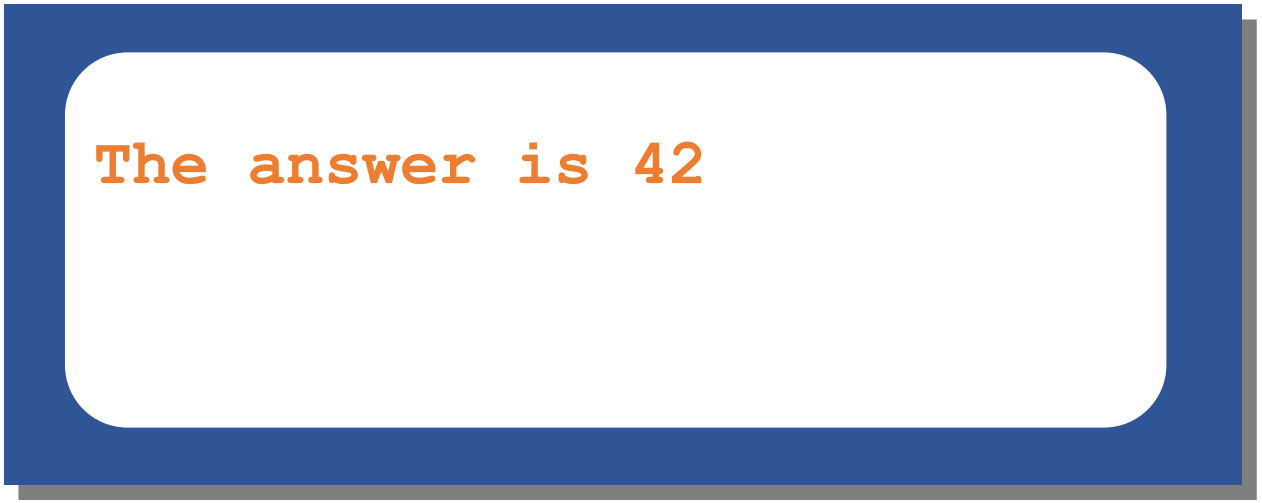
- Two strings are ***concatenated*** using the **+** operator.

```
String greeting = "Hello";
String sentence;
sentence = greeting + " officer";
System.out.println(sentence);
```

- Any number of strings can be concatenated using the **+** operator.

Concatenating Strings and Integers

```
String solution;
solution = "The answer is " + 42;
System.out.println (solution);
```



String Methods

- An object of the **`String`** class stores data consisting of a sequence of characters.
- Objects have methods as well as data
- The **`length()`** method returns the number of characters in a particular **`String`** object.

```
String greeting = "Hello";
int n = greeting.length();
```

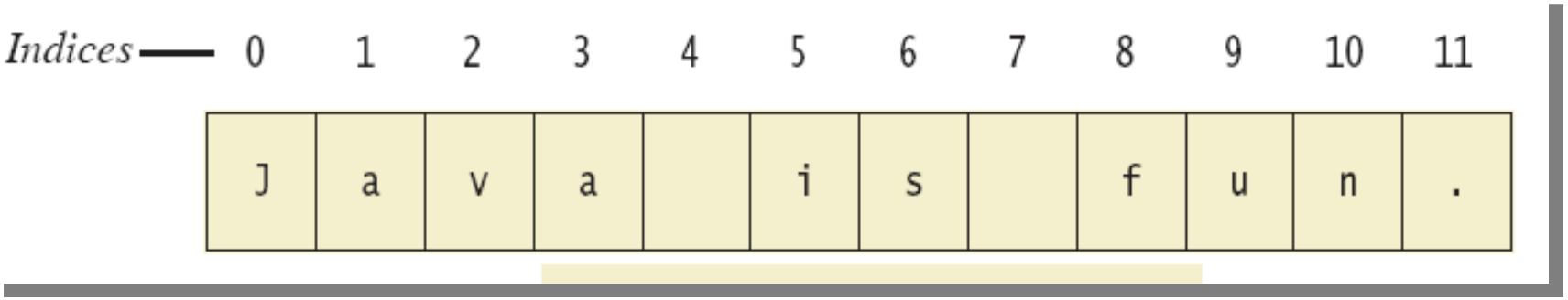
The Method `length()`

- The method `length()` returns an `int`.
- You can use a call to method `length()` anywhere an `int` can be used.

```
int count = command.length();
System.out.println("Length is " +
   command.length());
count = command.length() + 3;
```

`String` Indices

- Figure 2.4



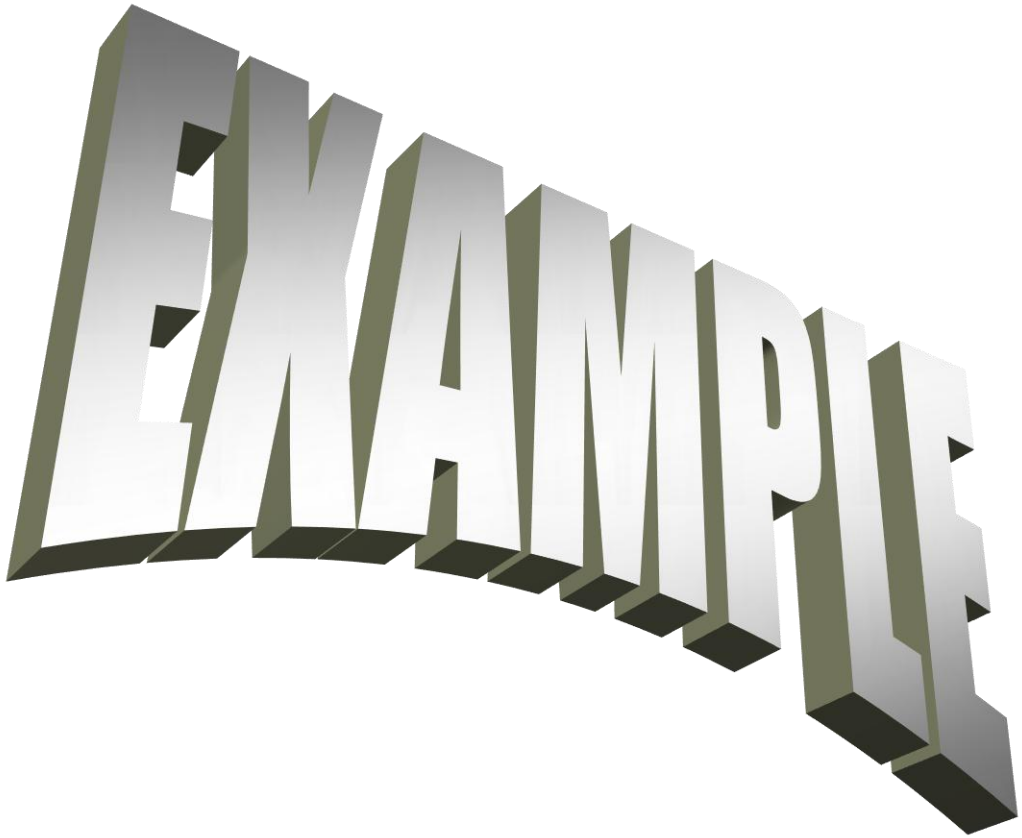
- Positions start with 0, not 1.
  - The 'J' in "Java is fun." is in position 0
- A position is referred to an an *index.*
  - The `'f'` in `"Java is fun."` is at index 8.

FIGURE 2.5 Some Methods in the Class `String`

Method	Return Type	Example for `String s = "Java";`	Description
`charAt` (*index*)	`char`	`c = s.charAt(2);` `// c='v'`	Returns the character at *index* in the string. Index numbers begin at 0.
`compareTo` (a_*string*)	`int`	`i = s.compareTo("C++");` `// i is positive`	Compares this string with *a_string* to see which comes first in lexicographic (alphabetic, with upper before lower case) ordering. Returns a negative integer if this string is first, zero if the two strings are equal, and a positive integer if *a_string* is first.
`concat` (a_*string*)	`String`	`s2 = s.concat("rocks");` `// s2 = "Javarocks"`	Returns a new string with this string concatenated with *a_string*. You can use the + operator instead.
`equals` (a_*string*)	`boolean`	`b = s.equals("Java");` `// b = true`	Returns true if this string and *a_string* are equal. Otherwise returns false.
`equals IgnoreCase` (a_*string*)	`boolean`	`b = s.equals("Java");` `// b = true`	Returns true if this string and *a_string* are equal, considering upper and lower case versions of a letter to be the same. Otherwise returns false.
`indexOf` (a_*string*)	`int`	`i = s.indexOf("va");` `// i = 2`	Returns the index of the first occurrence of the substring *a_string* within this string or -1 if *a_string* is not found. Index numbers begin at 0.

`lastIndexOf` (a_*string*)	`int`	`i = s.lastIndexOf("a");` `// i = 3`	Returns the index of the last occurrence of the substring *a_string* within this string or -1 if *a_string* is not found. Index numbers begin at 0.
`length()`	`int`	`i = s.length();` `// i = 4`	Returns the length of this string.
`toLower Case()`	`String`	`s2 = s.toLowerCase();` `// s = "java"`	Returns a new string having the same characters as this string, but with any uppercase letters converted to lowercase. This string is unchanged.
`toUpper Case()`	`String`	`s2 = s.toUpperCase();` `// s2 = "JAVA"`	Returns a new string having the same characters as this string, but with any lowercase letters converted to uppercase. This string is unchanged.
`replace` (*oldchar, newchar)*	`String`	`s2 = s.replace('a','o');` `// s2 = "Jovo";`	Returns a new string having the same characters as this string, but with each occurrence of *oldchar* replaced by *newchar*.
`substring` (*start)*	`String`	`s2 = s.substring(2);` `// s2 = "va";`	Returns a new string having the same characters as the substring that begins at index *start* through to the end of the string. Index numbers begin at 0.
`substring` (*start,end*)	`String`	`s2 = s.substring(1,3);` `// s2 = "av";`	Returns a new string having the same characters as the substring that begins at index *start* through to but not including the character at index *end*. Index numbers begin at 0.

`trim( )`	`String`	`s = "  Java  ";`
		`s2 = s.trim();`
		`// s2 = "Java"`
			Returns a new string having the same characters as this string, but with leading and trailing whitespace removed.



Example in more detail

`input =`

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

Current Line of Code

```
String input = keyboard.nextLine();
```

Example in more detail

input =

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

copyInput =

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

Current Line of Code

```
String copyInput = input;
```

Example in more detail

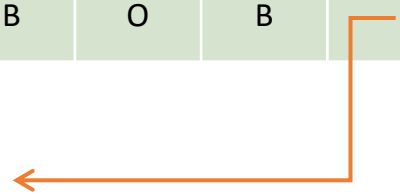
input =

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

copyInput =

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

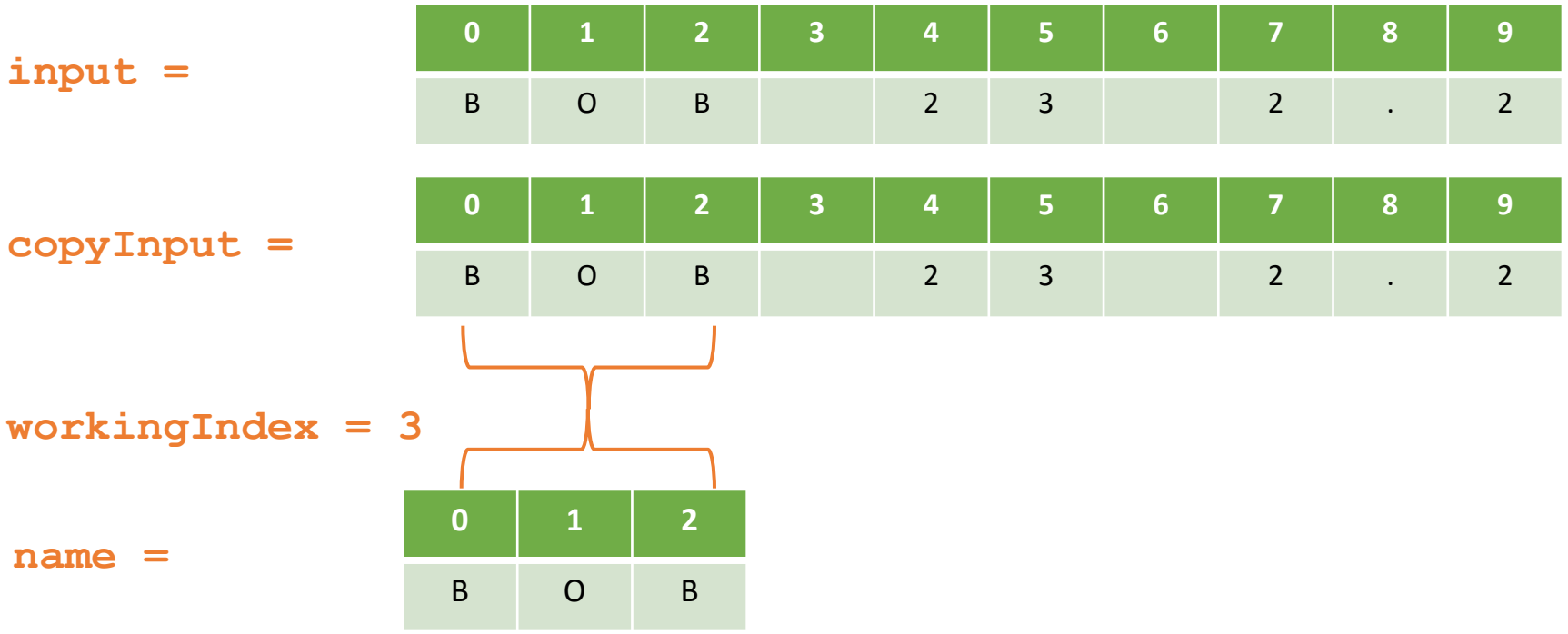
workingIndex = 3



Current Line of Code

```
int workingIndex = copyInput.indexOf(" ");
```

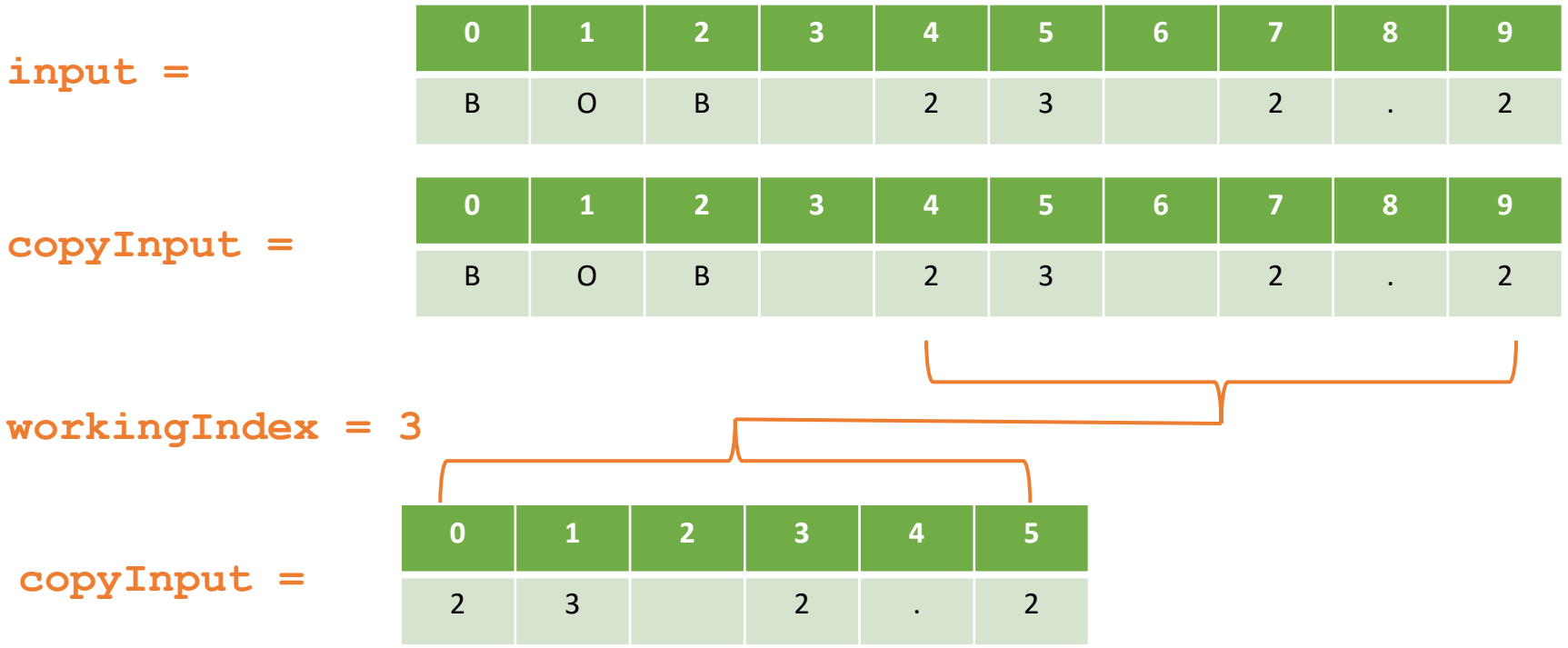
Example in more detail



Current Line of Code

```
String name = copyInput.substring(0,workingIndex);
```

Example in more detail



Current Line of Code

```
copyInput = copyInput.substring(workingIndex+1);
```

Example in more detail

input =

0	1	2	3	4	5	6	7	8	9
B	O	B		2	3		2	.	2

copyInput =

0	1	2	3	4	5
2	3		2	.	2

workingIndex = 3

Current Line of Code

```
copyInput = copyInput.substring(workingIndex+1);
```

Escape Characters

- How would you print

  `"Java" refers to a language.` ?

- The compiler needs to be told that the quotation marks (`"`) do not signal the start or end of a string, but instead are to be printed.

```
System.out.println(
"\"Java\" refers to a language.");
```

Escape Characters

\"	Double quote.
\'	Single quote.
\\	Backslash.
\n	New line. Go to the beginning of the next line.
\r	Carriage return. Go to the beginning of the current line.
\t	Tab. Add whitespace up to the next tab stop.

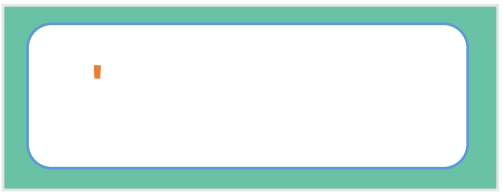
- Figure 2.6
- Each escape sequence is a single character even though it is written with two symbols.

Examples

```
System.out.println("abc\\def");
```


```
System.out.println("new\nline");
```


```
char singleQuote = '\'';
System.out.println
  (singleQuote);
```



The Unicode Character Set

- Most programming languages use the ***ASCII*** character set.
- Java uses the ***Unicode*** character set which includes the ASCII character set.
- The Unicode character set includes characters from many different alphabets (but you probably won't use them).

Keyboard and Screen I/O: Outline

- Screen Output
- Keyboard Input

Screen Output

- We've seen several examples of screen output already.
- `System.out` is an object that is part of Java.
- `println()` is one of the methods available to the `System.out` object.

Screen Output

- The concatenation operator (+) is useful when everything does not fit on one line.

```
System.out.println("Lucky number = " + 13 +
  "Secret number = " + number);
```

- Do not break the line except immediately before or after the concatenation operator (+).

Screen Output

- Alternatively, use `print()`

```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println(" shut the door.");
```

ending with a `println()`.

Keyboard Input

- Java has reasonable facilities for handling keyboard input.
- These facilities are provided by the `Scanner` class in the `java.util` package.
  - A *package* is a library of classes.

Using the Scanner Class

- Near the beginning of your program, insert

```
import java.util.Scanner;
```

- Create an object of the **`Scanner`** class

```
Scanner keyboard =
   new Scanner (System.in)
```

- Read data (an **`int`** or a **`double`**, for example)

```
int n1 = keyboard.nextInt();
double d1 = keyboard,nextDouble();
```

Some `Scanner` Class Methods

- Figure 2.7a

Scannner_Object_Name`.next()`
Returns the `String` value consisting of the next keyboard characters up to, but not including, the first delimiter character. The default delimiters are whitespace characters.

Scannner_Object_Name`.nextLine()`
Reads the rest of the current keyboard input line and returns the characters read as a value of type `String`. Note that the line terminator `'\n'` is read and discarded; it is not included in the string returned.

Scannner_Object_Name`.nextInt()`
Returns the next keyboard input as a value of type `int`.

Scannner_Object_Name`.nextDouble()`
Returns the next keyboard input as a value of type `double`.

Scannner_Object_Name`.nextFloat()`
Returns the next keyboard input as a value of type `float`.

Some `Scanner` Class Methods

- Figure 2.7b

Scannner_Object_Name`.nextLong()`
Returns the next keyboard input as a value of type `long`.

Scannner_Object_Name`.nextByte()`
Returns the next keyboard input as a value of type `byte`.

Scannner_Object_Name`.nextShort()`
Returns the next keyboard input as a value of type `short`.

Scannner_Object_Name`.nextBoolean()`
Returns the next keyboard input as a value of type `boolean`. The values of `true` and `false` are entered as the words *true* and *false*. Any combination of uppercase and lowercase letters is allowed in spelling *true* and *false*.

Scannner_Object_Name`.useDelimiter(`*Delimiter_Word*`);`
Makes the string *Delimiter_Word* the only delimiter used to separate input. Only the exact word will be a delimiter. In particular, blanks, line breaks, and other whitespace will no longer be delimiters unless they are a part of *Delimiter_Word*.

This is a simple case of the use of the `useDelimiter` method. There are many ways to set the delimiters to various combinations of characters and words, but we will not go into them in this book.

`nextLine()` Method Caution

- The `nextLine()` method reads
  - The remainder of the current line,
  - Even if it is empty.

`nextLine()` Method Caution

- Example – given following declaration.

```
int n;
String s1, s2;
n = keyboard.nextInt();
s1 = keyboard.nextLine();
s2 = keyboard.nextLine();
```

- Assume input shown

`n` is set to `42`

but `s1` is set to the empty string.

```
42
and don't you
forget it.
```

The Empty String

- A string can have any number of characters, including zero.
- The string with zero characters is called the *empty* string.
- The empty string is useful and can be created in many ways including

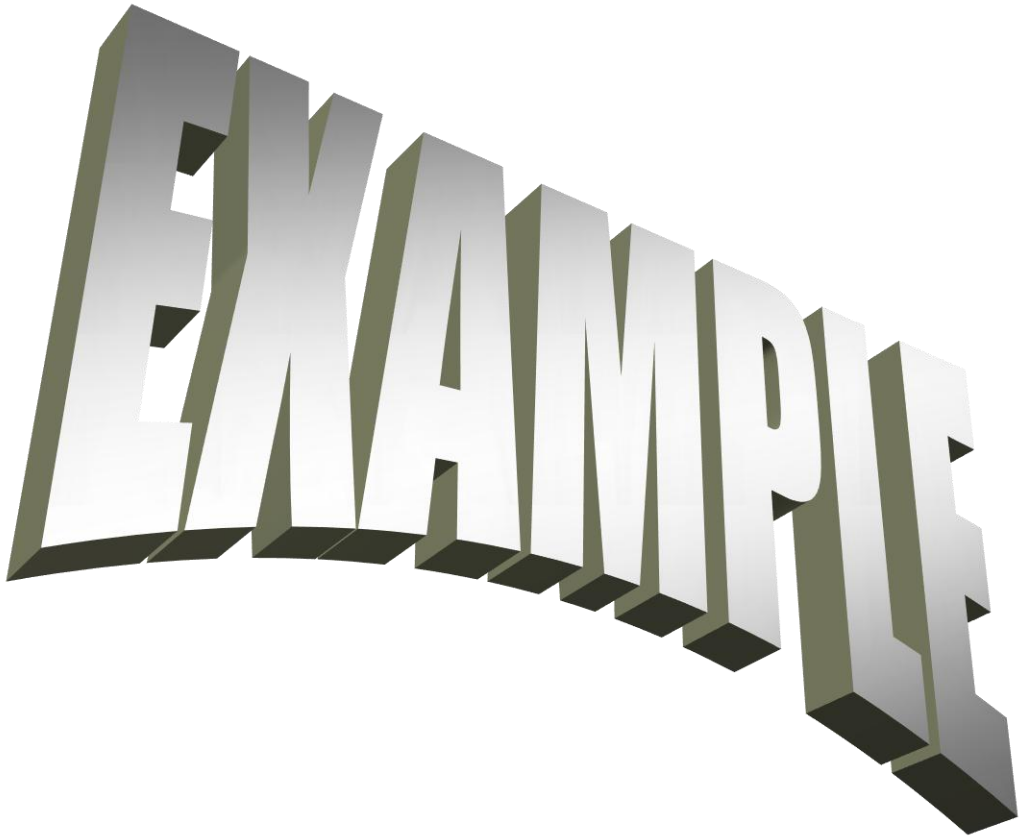
```
String s3 = "";
```

Other Input Delimiters (optional)

- Almost any combination of characters and strings can be used to separate keyboard input.
- to change the delimiter to `"##"`

```
keyboard2.useDelimiter("##");
```

  - whitespace will no longer be a delimiter for `keyboard2` input



Documentation and Style: Outline

- Meaningful Names
- Comments
- Indentation
- Named Constants

Documentation and Style

- Most programs are modified over time to respond to new requirements.
- Programs which are easy to read and understand are easy to modify.
- Even if it will be used only once, you have to read it in order to debug it .

Meaningful Variable Names

- A variable's name should suggest its use.
- Observe conventions in choosing names for variables.
  - Use only letters and digits.
  - "Punctuate" using uppercase letters at word boundaries (e.g. `taxRate`) – called “Camel Case” or “camelCase” or…
  - Start variables with lowercase letters.
  - Start class names with uppercase letters.

Comments

- The best programs are self-documenting.
  - Clean style
  - Well-chosen names
- Comments are written into a program as needed explain the program.
  - They are useful to the programmer, but they are ignored by the compiler.

Comments

- A comment can begin with //.
- Everything after these symbols and to the end of the line is treated as a comment and is ignored by the compiler.

```
double radius; //in centimeters
```

Comments

- A comment can begin with `/*` and end with `*/`
- Everything between these symbols is treated as a comment and is ignored by the compiler.

```
/**
This program should only
be used on alternate Thursdays,
except during leap years, when it should
only be used on alternate Tuesdays.
*/
```

Comments

- A *javadoc* comment, begins with `/**` and ends with `*/`.
- It can be extracted automatically from Java software.

```
/** method change requires the number of coins to be
  nonnegative */
```

When to Use Comments

- Begin each program file with an explanatory comment
  - What the program does
  - The name of the author
  - Contact information for the author
  - Date of the last modification.
- Provide only those comments which the expected reader of the program file will need in order to understand it.

Indentation

- Indentation should communicate nesting clearly.
- A good choice is four spaces for each level of indentation.
- Indentation should be consistent.
- Indentation should be used for second and subsequent lines of statements which do not fit on a single line.

Indentation

- Indentation does not change the behavior of the program (not true in all languages, e.g. Python)
- Proper indentation helps communicate to the human reader the nested structures of the program

Summary

- You have become familiar with Java primitive types (numbers, characters, etc.).
- You have learned about assignment statements and expressions.
- You have learned about stings.
- You have become familiar with classes, methods, and objects.

Summary

- You have learned about simple keyboard input and screen output.