Sheridan

PROG 10082

Object Oriented Programming 1

Recap

Variable Initialization

- Sometimes when we declare a variable, we must give it a value.
- This is true of variables that we create inside a main() method or other method.
- These are called "local variables" and must be initialized with a value before we can use them.
- Usually we would just use a null value such as 0 (for numbers), "" (the null string, which is used for strings) or '\0' (for the char type).
- Otherwise, you can always use whatever initial value you require for your program.

To initialize a variable in its declaration statement, we could do the following:

```
String strFirstName = "Kaluha";

double dblSalary = 0.0;

int intNumRecords = 0;

String strSearchKey = "";
```

Important!
Note that we prefer to use 0.0 as the floating-point literal value for 0.
Why do we use 0.0 and not 0?
Because 0 is considered an integer, and 0.0 is considered a floating-point.

When you assign 0 to a floating-point variable (such as float or double),
the computer has to do the extra work of converting the 0 into 0.0.
You can make your program run more efficiently if you eliminate this extra
step by using 0.0 as a floating-point literal value for 0.

You can declare and/or initialize multiple variables of the same type in one statement. Each of the following is valid:

```
String strFirstName, strLastName;

int intXVal=0, intYVal=0, intZVal=0;

char chrInitial = 'S', chrKey;
```

Note:
We often don't declare and/or initialize variables on the same line because they're harder to read, and harder to document.

Exercises

- For each of the following statements below, declare a variable and initialize it to a null value. Use the most appropriate data types and identifier names.

  a. to store a customer's last name

  b. to record the number of customers

  c. to record the customer's outstanding balance

  d. to store the customer's phone number

  e: to record a single keystroke made by the user

- **Solutions:**

a. to store a customer's last name

```
String custLastName = "";
```

b. to record the number of customers

```
int numCustomers = 0;
```

c. to record the customer's outstanding balance

```
double custBalance = 0.0;
```

d. to store the customer's phone number

```
String custPhone = "";
```

e: to record a single keystroke made by the user

```
char keystroke = '\0';
```

Assignment Statements

An **expression** is a statement that involves a calculation using literals, variables, and operators; an expression can be evaluated into a result.

An **assignment statement** (or assignment expression) assigns a value/result to a variable. For example, each of these are assignment statements:

```
strFirstName = "Fred";

dblArea = 15.5 * 10.0;
```

When looking at an assignment statement, the equals sign (=) is the assignment operator, and it's job is to place the result of the expression on the right into the variable on the left.

It's important to note that the direction of assignment goes from right to left, not left to right. If it helps, read the assignment operator as "gets" or "gets the value/result of".

For example:

```
strFirstName = "Fred";
```

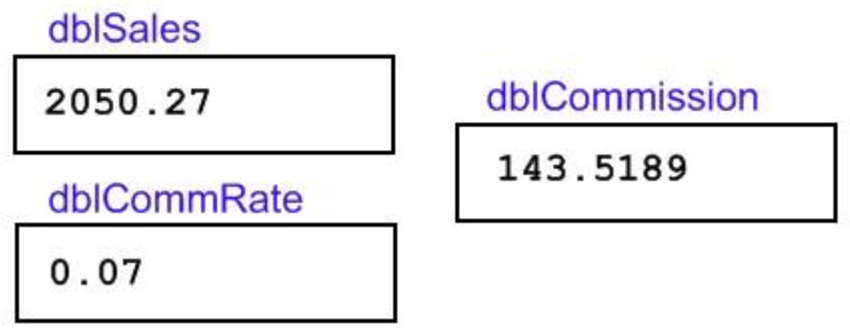
reads "the variable strFirstName gets the value Fred“

```
dblArea = 15.5 * 10.0;
```

reads "the variable dblArea gets the result of the expression 15.5 * 10.0"

Examine the following code segment and the diagram below:

```
double dblSales = 2050.27;
double dblCommRate = .07;
double dblCommission = dblSales * dblCommRate;
```



All three variables in memory.

Each variable is a location in memory, and each memory location or variable is given a value to store.

The first two statements store **literal values** or **hard-coded values** into the variables dblSales and dblCommRate.

The third statement calculates the value of dblSales multiplied by dblCommRate, and stores the result in the variable dblCommission.

Exercise

Examine the program and its output in the listing below and answer the following questions:

1. How many variables are used in this program?
2. What are the names of the variables?
3. Draw a diagram similar to the one in slide 10 that shows the values in each of the variables.

GradeCalculator.java

```
public class  GradeCalculator
{
    public static void main(String[] args)
    {
        int intAssignMark = 40;
        double dblActualMark = 32.5;
        int intBonus = 2;

        double dblPercent = (dblActualMark + intBonus) /
            intAssignMark * 100;

        System.out.println("Assignment mark: " + dblActualMark +
            "/" + intAssignMark + " (+" + intBonus + " bonus)");
        System.out.println("(" + dblPercent + "%)");
    }
}
```

The program's output:

```
Assignment mark: 32.5/40 (+2 bonus) (86.25%)
```

Solution:

1.There are 4 variables used in this program.

2. The 4 variables' names are: intAssignMark, dblActualMark, intBonus, and dblPercent.

3. Your prof will do this during clas.

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;

$$\xrightarrow{\texttt{range increases}}$$

`byte, short, int, long, float, double`

Casting in an Augmented Expression

In Java, an augmented expression of the form **x1 op= x2** is implemented as **x1 = (T)(x1 op x2)**, where **T** is the type for **x1**. Therefore, the following code is correct.

int sum = **0**;

sum += **4.5**; // sum becomes 4 after this statement

sum += 4.5 is equivalent to **sum = (int)(sum + 4.5)**.

Constants

- When programming, you will often need to use fixed or pre-defined values like the GST and PST rates, PI, limits and sentinel values, and other kinds of literals.
- It's generally considered bad style to write these values as literals right in the code, because it makes the program harder to read and harder to modify. Instead, we use special variables called **constants** to store special, fixed values during the run of a program.
- A constant is a special variable whose value never changes while a program is running.
- For example, a program that calculates and displays a customer bill for some window blinds will require the GST and PST rates when the tax amounts are calculated. GST and PST will not change while the program is running, and in fact they could be used over and over again throughout the program.
- Another example could be when calculating areas and volumes for circles, spheres, and cylinders -- you will require the value of PI in order to perform these calculations.

In a large program, using literals results in a huge waste of time when modifications are necessary.

Imagine a large program that uses the GST and PST rates in many lines of code.

If the tax rate actually changed some day, the programmer would have to search through thousands of lines of code and replace the old value with the new value.

Most editor's have a search-and-replace function, but this is not always feasible. What if we wanted to replace the PST of .08 with a new rate of .05 (wishfull thinking!) but there were other occurrences of the literal .08 in our code that had nothing to do with taxes!

We could do a search-replace for .08, but we'd have to monitor the operation, ensuring we didn't replace the occurrences of .08 that weren't tax rates.

A constant is defined and initialized with its value almost like a regular variable.

You must place the "**final**" keyword before the data type when defining a variable as a constant:

```
final double HST_RATE = .13;
```

Note the following:
•A constant name is always in upper-case letters.
•If multiple words are used in a constant name, they are always separated with the underscore (_) character.

•Assign the value to the constant when it is declared, as they are not allowed to be changed in code.

In your textbook (section 2.7), they define constants inside the main method.

In reality, constants are almost always defined at the top of the class outside the main method, between the public class statement and the main method header.

Because this is a different area of your program that has some meaning to Java, we have to add the keywords **public static** to our constant declarations:

```
public class Example{
    public static final double HST_RATE = .13;

    public static void main(String[] args)
    {
        // ...
    }
}
```

Once you've defined your constant, you simply refer to the constant by name (e.g. HST_RATE) in your code instead of using the literal:

```
public class Example
{
   public static final double HST_RATE = .13;

   public static void main(String[] args)
   {
      // imagine other code here
      // .....
      double taxAmt = subTotal * HST_RATE;
   }
}
```

If the HST rate changes, you only need to change the one statement that defines the constant!

Exercises

Create a new program and define a constant called COLLEGE_NAME. Give the constant the value "Sheridan College".

In your main() method, add the following statements:

```
System.out.println("You attend " + COLLEGE_NAME + ".");
COLLEGE_NAME = "Sheridan College Institute of Technology and Advanced Learning";
System.out.println("You should really call it " + COLLEGE_NAME);
```

What happens when you compile this program? Why?

2. Edit your solutions to programming questions 2.1 to 2.7 at the end of Chapter 2 and be sure to use constants where applicable.

Solution:

1- The code

```
public class ConstTest {
// constant for the name of the college
public static final String COLLEGE_NAME = "Sheridan College";

public static void main(String[] args) {
 // print out where we work/attend school
System.out.println("You attend " + COLLEGE_NAME + ".");
// change the name! o.O NOPE
COLLEGE_NAME = "Sheridan College Institute of Technology and Advanced Learning";
// print it out again!
System.out.println("You should really call it " + COLLEGE_NAME);
   }
 }
```

You should get an error when you compile

("Error: cannot assign a value to final variable COLLEGE_NAME").

The values of constant variables can never change once they're declared and assigned!

- **Shorthand Assignment Operators**
- There are some special assignment operators you can use as shortcuts to certain kinds of assignment statements.

```
public class  AssignOps {
  public static void main(String[] args) {


    int sum = 10;


    System.out.println("The value stored in sum is "  + sum);
    sum = sum + 20;
    System.out.println("The value now stored in sum is "  + sum);
  }
}
```

This small program demonstrates a situation where you can use the special += assignment operator.

This operator, the "plus-equals" operator, adds the operand on the right to the operand on the left.

If you rewrote the program using the += operator, you'd have:

```
public class  AssignOps {
  public static void main(String[] args) {

    int sum = 10;

    System.out.println("The value stored in sum is "  + sum);
    sum += 20;
    System.out.println("The value now stored in sum is "  + sum);
  }
}
```

There are also similar operators to perform subtraction, division, and modulus.

Exercises

- Determine, without coding a program, the output of the following code segment:

```
int  counter = 1;

int  increment = 2;
System.out.print(counter + " ");

counter += increment;

System.out.print(counter + " ");

counter *= increment;

System.out.print(counter + " ");

increment /= 2;

counter -= increment;

System.out.println(counter);

System.out.println("increment: "  + increment);
```

- Solution:

Output:

```
1 3 6 5
increment: 1
```

Unary Operators

- In addition to the assignment operators, there are also some **unary** operators.
- Recall that regular operators, such as
- +, * and += are **binary** operators - they require two operands, one on each side.
- Unary operators require only 1 operand
- Unary operators in Java can go on the left or the right of the operand.
- The difference is in when the increment actually takes place. Some of the exercises demonstrate this.

The code listing below demonstrates how the increment operator works:

```
public  class  UnaryOps
{
  public  static  void  main(String[] args)
  {
    int  intNum1 = 2;
    int  intNum2 = 0;

    System.out.print("intNum1:  "  + intNum1);
    System.out.println("  intNum2:  "  + intNum2);
    System.out.println();

    intNum2++;
    System.out.println("increment intNum2:  "  + intNum2);
    intNum2++;
    System.out.println("increment intNum2:  "  + intNum2);
    intNum2++;
    System.out.println("increment intNum2:  "  + intNum2);

    intNum2 += intNum1;
    System.out.println();
    System.out.print("intNum1:  "  + intNum1);
    System.out.println("  intNum2:  "  + intNum2);

  }
}
```

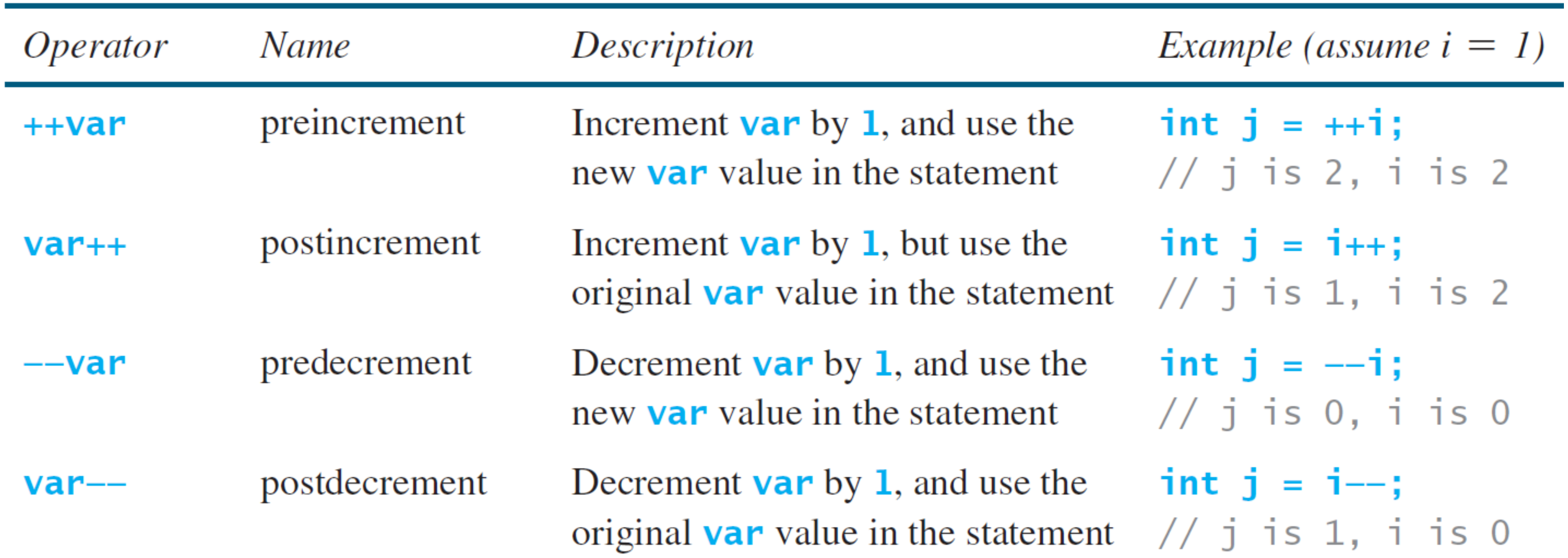
The output for this program:

```
intNum1: 2 intNum2: 0

increment intNum2: 1
increment intNum2: 2
increment intNum2: 3
intNum1: 2 intNum2: 5
```

Augmented Assignment Operators

Operator	*Name*	*Example*	*Equivalent*
`+=`	Addition assignment	`i += 8`	`i = i + 8`
`-=`	Subtraction assignment	`i -= 8`	`i = i - 8`
`*=`	Multiplication assignment	`i *= 8`	`i = i * 8`
`/=`	Division assignment	`i /= 8`	`i = i / 8`
`%=`	Remainder assignment	`i %= 8`	`i = i % 8`

Increment and Decrement Operators



Operator	*Name*	*Description*	*Example (assume i = 1)*
`++var`	preincrement	Increment `var` by `1`, and use the new `var` value in the statement	`int j = ++i;` `// j is 2, i is 2`
`var++`	postincrement	Increment `var` by `1`, but use the original `var` value in the statement	`int j = i++;` `// j is 1, i is 2`
`--var`	predecrement	Decrement `var` by `1`, and use the new `var` value in the statement	`int j = --i;` `// j is 0, i is 0`
`var--`	postdecrement	Decrement `var` by `1`, and use the original `var` value in the statement	`int j = i--;` `// j is 1, i is 0`

The code listing below demonstrates how the increment operator works:

```
public  class  UnaryOps
{
  public  static  void  main(String[] args)
  {
    int  var1 = 1;
    int var2 = 1;

    System.out.println(“Var1:  " + (var1++));
      System.out.println(“Var1:  " + (var1));

     System.out.println();

    System.out.println(" Var2:  " + (++var2));
     System.out.println(" Var2:  " + (var2));
    System.out.println();


  }
}
```

The output for this program:

```
Var1 : 1
Var1 : 2

Var2 : 2
Var2 : 2
```

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--;
```

Practice

Given that x=5, y=2, and z=10, what do you think would be the value of "result" after the following assignment statements?

```
result = ++x + y;
```

```
result = z/y; result += x * y;
```

```
result = x++ + y;
```

For each of the examples above, type the code into a Java program and print/display the value of "result". Do you get the output you expected? Why or why not?

When the ++ or -- operator is to the left of the operand, we refer to it as a **pre-fix operator**.

When the ++ or -- operator is to the right of the operand, we call it the **post-fix operator**.

The pre-fix operator has higher precedence than the post-fix operator.

In a statement with multiple types of operators, the pre-fix operator will execute before most other operators.

The post-fix operator will usually execute after the other operators.

Character

- As mentioned last time, the character data type represents a single character.
  - **char** letter = ‘A’;
  - **char** numChar = ‘4’;
- Characters are enclosed in single quotes, Strings in double quotes.
  - **String** aString = “A”;

Character encoding

- Computer language is binary.
- A character therefore is stored as a sequence of 1s and 0s.
- ***Encoding*** is the mapping of a character to its binary representation.
- More than one way to encode a character. ***Encoding scheme*** is the way that characters are encoded.

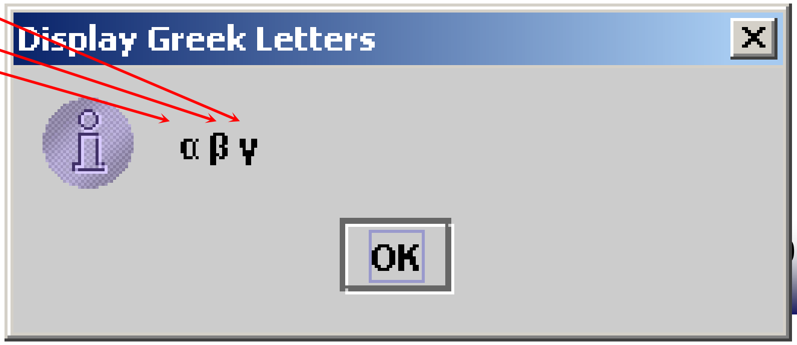
Unicode

- Java supports unicode.
- Unicode was designed to be able to hold characters from all the world’s languages.
- Initially 16-bit, but apparently not enough for all characters.
- Anything beyond the 16-bit limit are called supplementary characters.
- The 16-bit char can hold unicode characters

Unicode

- The 16-bit Unicode chars consist of 2 bytes preceded by \u and written in four hex digits.
- Range: '\u0000' to '\uFFFF' So, Unicode can represent `65535 + 1 characters`.

Unicode \u03b1 \u03b2 \u03b3 for three Greek letters



ASCII

- ASCII (American Standard Code for Information Interchange)
  - 7-bit coding scheme
  - Represents all uppercase, lowercase letters, digits, punctuation marks, and control characters.
- Unicode includes ASCII code from ‘\u0000’ to ‘\u007F’

ASCII and Unicode

- You can use both ASCII and Unicode in a program
  - E.g.
    char letter = ‘A’; //(ASCII)
  - is Equivalent to:
    char letter = ‘\u0041’; // ‘A’ in Unicode is 0041

    char numChar = '4'; (ASCII)

    char numChar = '\u0034'; (Unicode)

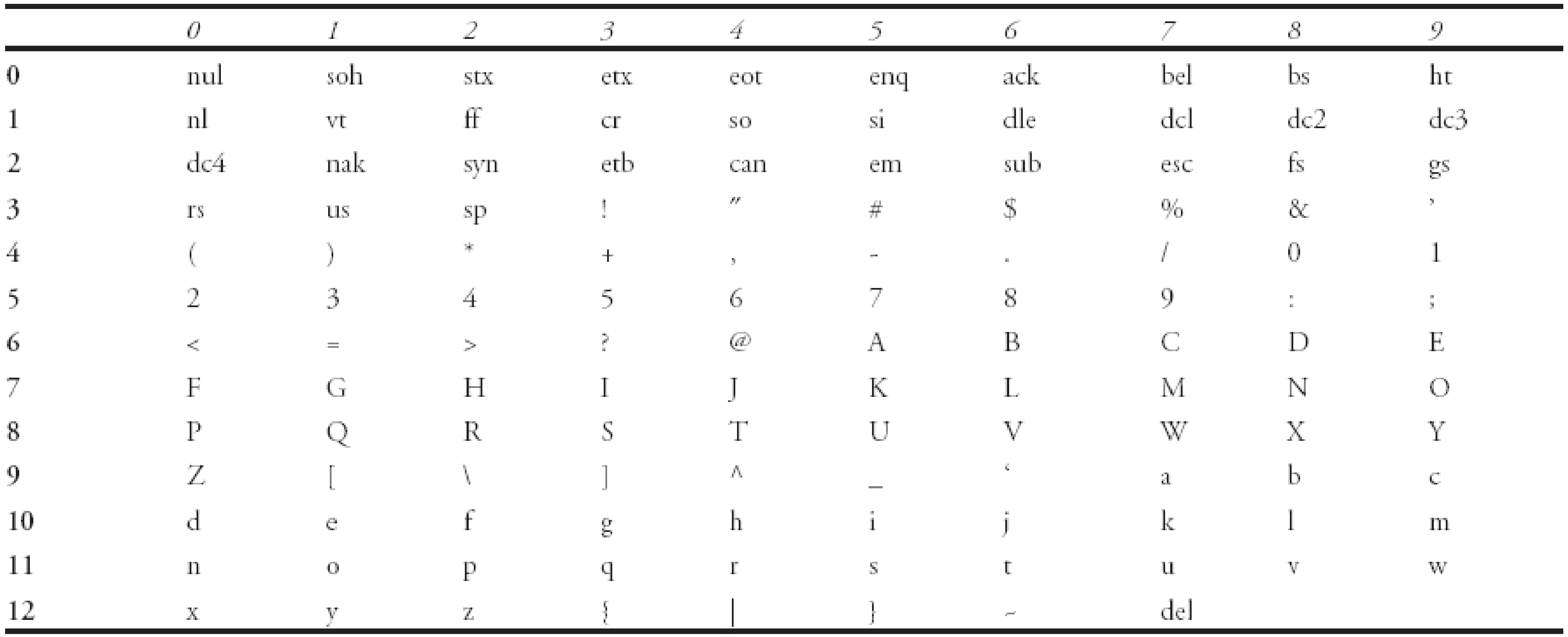
ASCII Code for Commonly Used Characters

Characters	Code Value in Decimal	Unicode Value
`'0'` to `'9'`	48 to 57	\u0030 to \u0039
`'A'` to `'Z'`	65 to 90	\u0041 to \u005A
`'a'` to `'z'`	97 to 122	\u0061 to \u007A

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

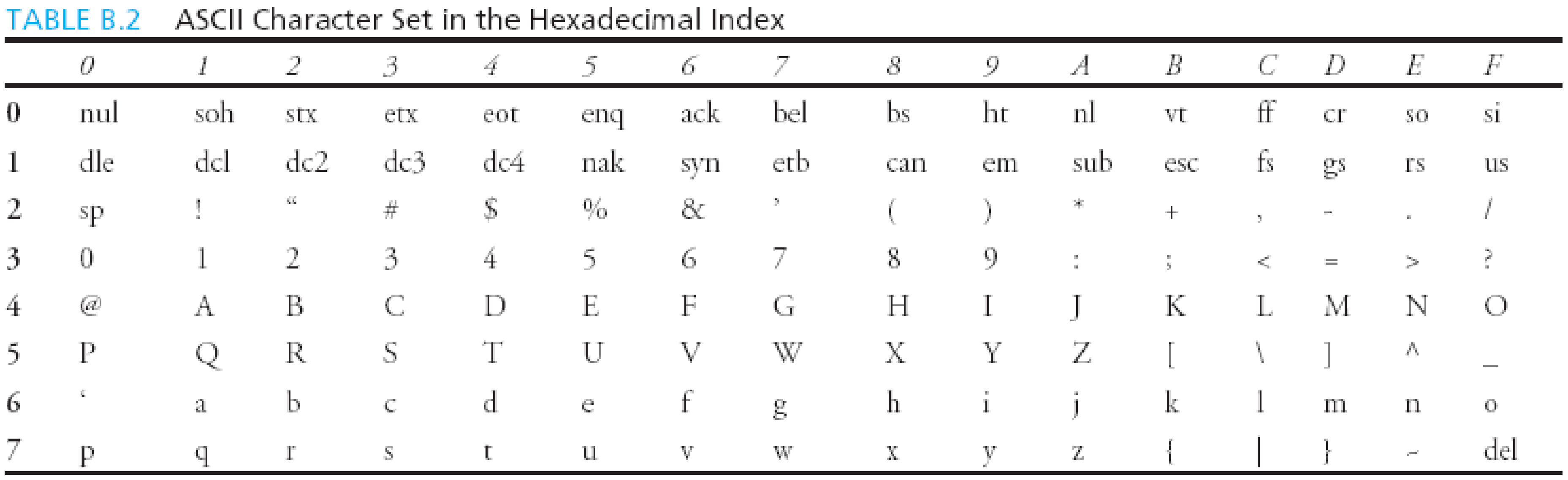


TABLE B.2 ASCII Character Set in the Hexadecimal Index

	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

Increment and decrement operators

- You can use the increment and decrement operators on characters.
- Returns the next or previous letter.
  - char ch = ‘a’;
    System.out.println(++ch);

Escape Sequences for Special Characters

Special characters need to be escaped.
From last class:
Some special characters that cannot be typed, so have to be escaped:

Escape Sequence	*Name*	*Unicode Code*	*Decimal Value*
`\b`	Backspace	`\u0008`	8
`\t`	Tab	`\u0009`	9
`\n`	Linefeed	`\u000A`	10
`\f`	Formfeed	`\u000C`	12
`\r`	Carriage Return	`\u000D`	13
`\\`	Backslash	`\u005C`	92
`\"`	Double Quote	`\u0022`	34

Casting

- Casting between types
- Casting involves loss of precision.
- Changes value from one type to another
  - E.g. Char ch = (char) 65.25;

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

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

Casting

- When we cast an int to char, only the lower 16bits of data are used.
  – E.g. Char ch = (char) 0xAB0041;
  – //The lower 16 bits hex code 0041 is assigned to ch. ch is character ‘A’
- When we cast a floating point value, it is first converted to int, and then cast to char.
  – E.g. Char ch = (char)65.25;
  – // decimal 65 is assigned to ch, and it gives the character A

Implicit casting

- Implicit casting can be used if the result of the casting fits in the variable.
  - E.g.	byte b = ‘a’;
  - Works because ‘a’ is 97 so it fits the 128 values range.
- Explicit casting
  - E.g.	byte b = ‘uFFF4’;
  - Does not work because value cannot fit into a byte.

Explicit casting

- If the value cannot fit in a byte you can force it:
- byte b = (byte)’uFFF4’;
- If the value cannot fit in a variable, it must be cast explicitly.

Numeric operations on chars

- Because char is represented similarly to int, you can apply all numeric operators on them.
- int i = 1 + ‘a’; //i = 98, ‘a’ = 97
- System.out.println((char) i); //This would print ‘b’.

Exercise

- Write a java program that assign your grade (B or C or D) to a character variable then print the following message on the screen:

My current grade is B

My plan is to get grade A

One level more ↙

- *Try to use casting and without casting.*

Solution

```
public class try1
{
   public static void main(String[] args)
   {
     char grade='B';
     System.out.println("my current grade is "+ grade);

     System.out.println("my plan is to get grade\t"+ (char)(grade-1));
   }
};
```

Character Operations

```
public class CharacterStuff {
public static void main(String[ ] args) {
   char c = 'a ‘;
   int intChar = c;
   System.out.println("char: " + c);
   System.out.println("int: " + intChar);
   c++;
   System.out.println(c);
    c -= 32;
   System.out.println(c);
     }
   }
```

Your output should appear as:

```
char: a int: 97 b B
```

```
public class CastingEx {
    public static void main(String[] args) {
        int num1 = 5;
        double num2 = 8.2;
        char letter = 'A';
        int ascE = 69;
      int num3 = num2;
      double num4 = num1;
      int anA = letter;
      char anE = ascE;

    System.out.println("num3: " + num3);
    System.out.println("num4: " + num4);
    System.out.println("anA: "+ anA);
    System.out.println("anE: "+ anE);
  }
}
```

When you compile this program, you'll receive the following two errors for statements 1 and 4 (your actual line numbers given by the compiler might vary):

```
TestingCasts.java:10: possible
loss of precision found     :
double
required: int int num3 = num2;

TestingCasts.java:13: possible
loss of precision found:
int  required: char
char anE = ascE;
```

Important!

In order to use the unary operators on a variable, that variable must be initialized! To understand why, compile the following code:

```
public class TestUnaryOps {
   public static void main(String[] args) {
      int x;
      x++;
      System.out.println(x);
   }
}
```

When you compile, you'll get the error message

```
TestUnaryOps.java:4: variable x might not have been initialized
      x++;
      ^
1 error
```

This error message is telling you that the variable x was declared, but was not initialized with a value. Unary operators and the special assignment operators (such as +=) do not work on variables that have not yet been initialized!

IPO

So far, we've written programs that display output on the console.

If we needed data in our program, we typed it in the source code.

It is more likely that most of your inputs will come from other source, such as a file or from the user.

You'll work with files in the next Java course, but in this course we can look at a few different ways to get user input.

IPO

- What does "IPO" stand for?
- Input-Processing-Output
- **IPO stands** for Input-Processing-Output:

Input consists of the data that goes into the program.

Most programs need to have data to work with, so as a programmer, you need to decide how the input gets into the program.

- **Processing** is what happens to the data inside the program. This is where a lot of your code will be involved, especially in larger programs.

  Sometimes there are many different kinds of processing tasks going on in the same program. Such tasks include calculating, manipulating text or numeric data, searching, sorting, and comparing.

- **Output** is the end result of your program, or the results of the processing. The inputs are transformed into outputs by the processing tasks that have been performed.

  The programmer also needs to decide what form the outputs will take. For example, are they displayed on the screen, printed with a printer, or saved to a file?

Example:

A program calculates the area of a room.

Possible inputs, processing, and outputs might be:

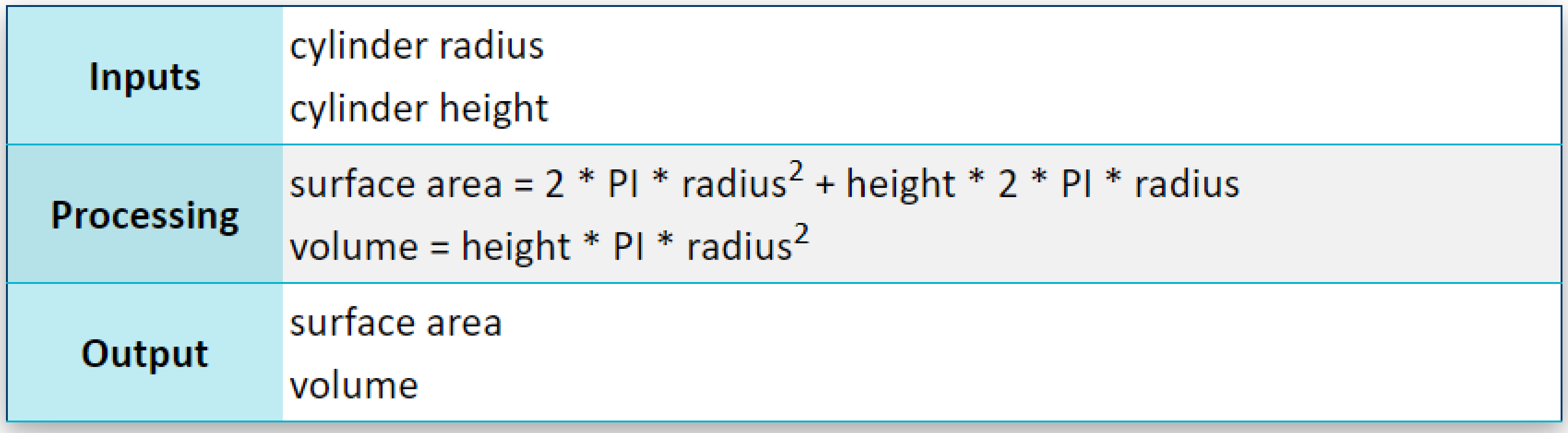
Inputs: length of room, width of room

Processing: area = length * width

Output: the area of the room

Example:

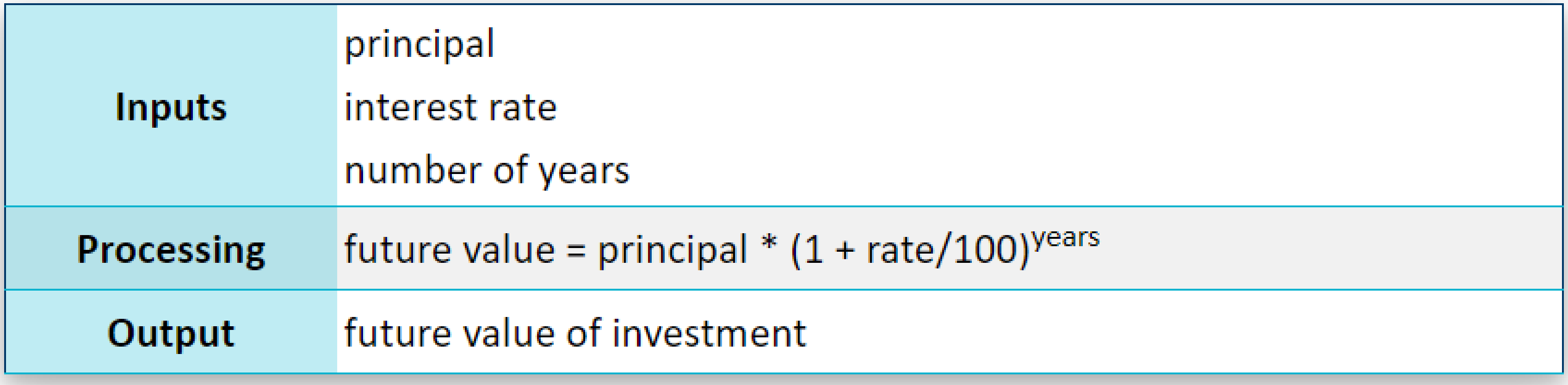
Define the inputs, processing, and outputs for a program that gives the surface area and volume of a cylinder.



Inputs	cylinder radius cylinder height
Processing	surface area = 2 * PI * radius^2 + height * 2 * PI * radius volume = height * PI * radius^2
Output	surface area volume

Example 2:

Define the inputs, processing, and outputs for a program that displays the future value of an investment for a principal P at a rate R compounded yearly for n years. The formula for compound interest is final amount = $P(1+R/100)^n$.



Inputs	principal interest rate number of years
Processing	future value = principal * $(1 + rate/100)^{years}$
Output	future value of investment

Reading Input from the Console

The Java Scanner class provides various methods to read and parse various primitive values.

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

A simple text scanner which can parse primitive types and strings using regular expressions. A Scanner breaks its input into tokens using a delimiter pattern, which by default matches whitespace. The resulting tokens may then be converted into values of different types using the various next methods.

For example, this code allows a user to read a number from System.in:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

There is a special class in Java called the Scanner class. So far we've been using classes like System and String.

These classes contain methods that we can use (like println() and print()).

The Scanner class is a different kind of class. You have to create a Scanner **object** in order to use the methods of the Scanner class. We do this by creating a variable, and instead of making the variable a primitive type (like int or double), we define it as "Scanner type":

```
Scanner input;
```

The variable here is called "input", and we've declared this variable to hold Scanner objects. The identifier "input" follows the same standards and conventions for any variable identifier, so remember that you can call this variable anything you like as long as it follows those same rules. For example, I could have said:

```
Scanner scanner;
```

Next, we have to construct the Scanner object and store it in the input variable:

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

In this statement, the **`new Scanner()`** part of the statement constructs a new Scanner object.

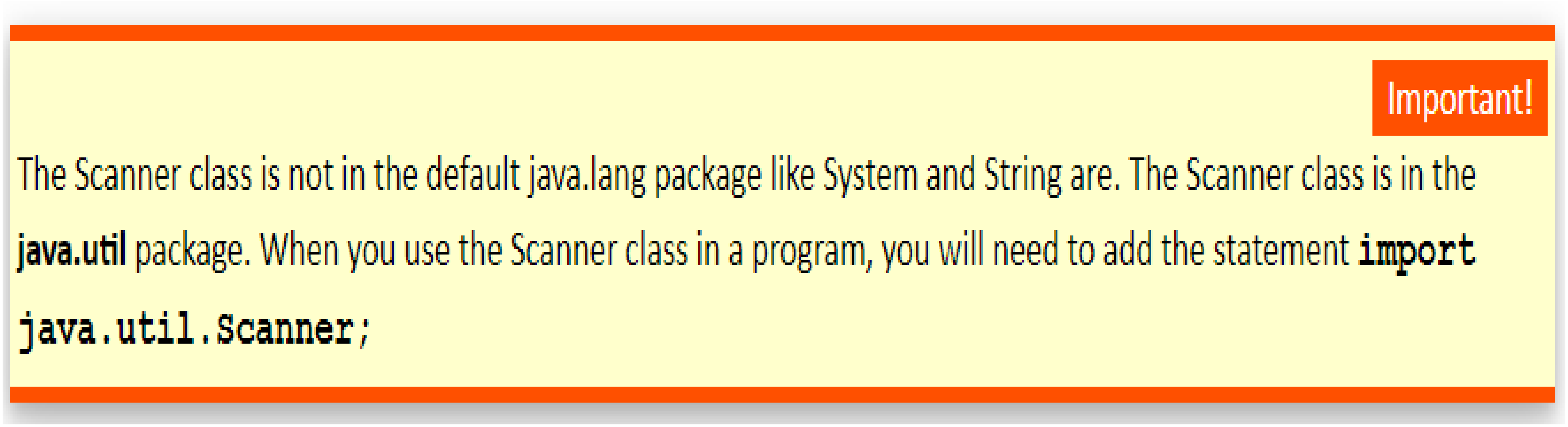
A Scanner object will read inputs for you, but it needs to know which device you want to read inputs from. We want it to read inputs from the keyboard, which is the default input device, so we tell it that we want it to use System.in.

We do this by saying **`new Scanner(System.in)`**

If you prefer, you can put this all on one statement, instead:

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

This statement says, "Create a new Scanner object that reads data from the keyboard, and then reference that Scanner object via the **`input`** variable." Once you've created a Scanner object, you can use many of the method that it contains.


Important!

The Scanner class is not in the default java.lang package like System and String are. The Scanner class is in the **java.util** package. When you use the Scanner class in a program, you will need to add the statement **`import java.util.Scanner;`**

The Scanner object's methods will read in data until it reaches any "whitespace" character or characters.

This includes spaces, tabs, and new-lines.

One problem with the scanner class is when you want to read an entire string with spaces.

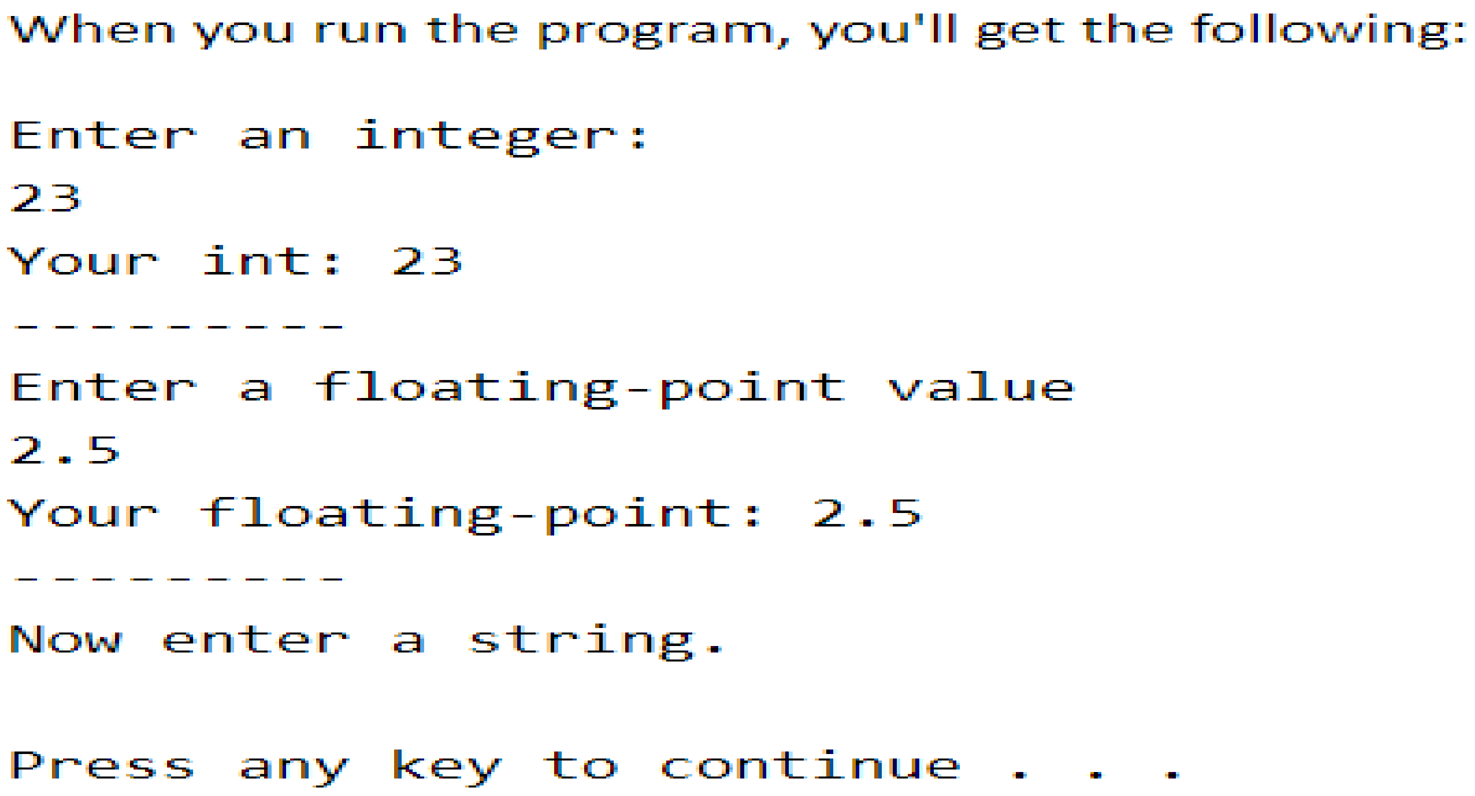
Reading Numbers from the Keyboard

```
Scanner input = new Scanner(System.in);
int value = input.nextInt();
```

Method	**Description**
`nextByte()`	reads an integer of the `byte` type.
`nextShort()`	reads an integer of the `short` type.
`nextInt()`	reads an integer of the `int` type.
`nextLong()`	reads an integer of the `long` type.
`nextFloat()`	reads a number of the `float` type.
`nextDouble()`	reads a number of the `double` type.

Try the following program to understand the problem:

```java
import java.util.Scanner;

public class  TryScanner {

    public static void main(String[] args) {

        Scanner input = new Scanner(System.in);
        System.out.println("Enter an integer:  ");
        int num = input.nextInt();
        System.out.println("Your int:  " + num);

        System.out.println("---------");
        System.out.println("Enter a floating-point value:");
        double num2 = input.nextDouble();
        System.out.println("Your floating-point:  " + num2);

        System.out.println("---------");
        System.out.println("Now enter a string.");
        String str = input.nextLine();
        System.out.println(str);
    }
}
```



When you run the program, you'll get the following:

```
Enter an integer:
23
Your int: 23
----------
Enter a floating-point value
2.5
Your floating-point: 2.5
----------
Now enter a string.

Press any key to continue . . .
```

When the program runs, the prompt to enter a string appears but there is no opportunity to enter anything!

What if we use the next() method instead of nextLine()? Change the last section of your code:

```
import java.util.Scanner;
public class TryScanner {
  public static void main(String[] args) {

    Scanner input = new Scanner(System.in);
     System.out.println("Enter an integer: ");
     int num = input.nextInt();
     System.out.println("Your int: " + num);
     System.out.println("---------");
     System.out.println("Enter a floating-point value:");
     double num2 = input.nextDouble();
     System.out.println("Your floating-point: " + num2);

     System.out.println("---------");
     System.out.println("Now enter a string.");
     String str = input.next();
     System.out.println(str);
        }
}
```

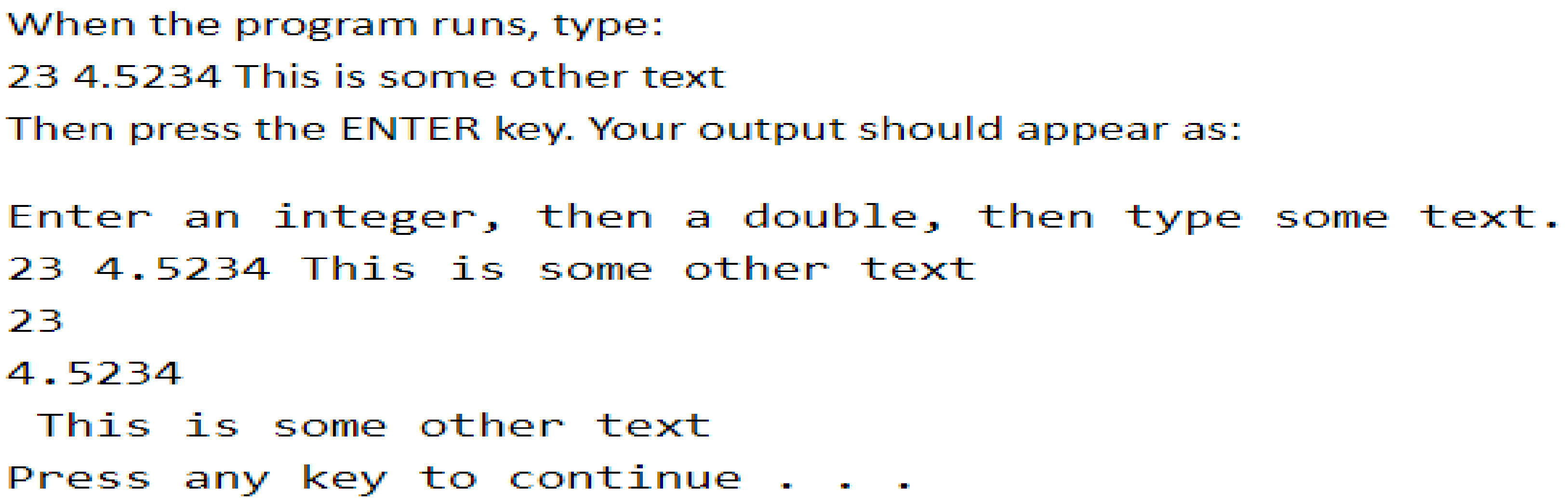
```
Enter an integer:
23
Your int: 23
---------
Enter a floating-point value
2.5
Your floating-point: 2.5
---------
Now enter a string.
Hi there!  How are you?
Hi
Press any key to continue . . .
```

This time, we were allowed to enter a string, but only the first word in the string was saved to the variable. That's because the next() method stops "recording" at the first space.

But why didn't nextLine() work? The nextLine() method is supposed to stop recording when it sees a new-line (enter key). The problem is not actually with nextLine(). The problem is actually the previous calls to Scanner's methods.

When a user types values using the keyboard, each keystroke is stored in the **input buffer**.
The input buffer is an area of memory that the Scanner object uses to "record" input values. Whenever you use one of the Scanner's methods, it goes to the input buffer and reads back the values stored there. As it reads a value, that value is removed from the input buffer.

```
import java.util.Scanner;
public class TryScanner3 {
        public static void main(String[] args) {
          Scanner keyIn = new Scanner(System.in);
           System.out.println("Enter an integer, then a " + "double, then type
                                        some text");
          int intNum = keyIn.nextInt();
         double dbNum = keyIn.nextDouble();
         String strText = keyIn.nextLine();

        System.out.println(intNum);
        System.out.println(dbNum);
        System.out.println(strText);
  }
}
```



When the program runs, type:

23 4.5234 This is some other text

Then press the ENTER key. Your output should appear as:

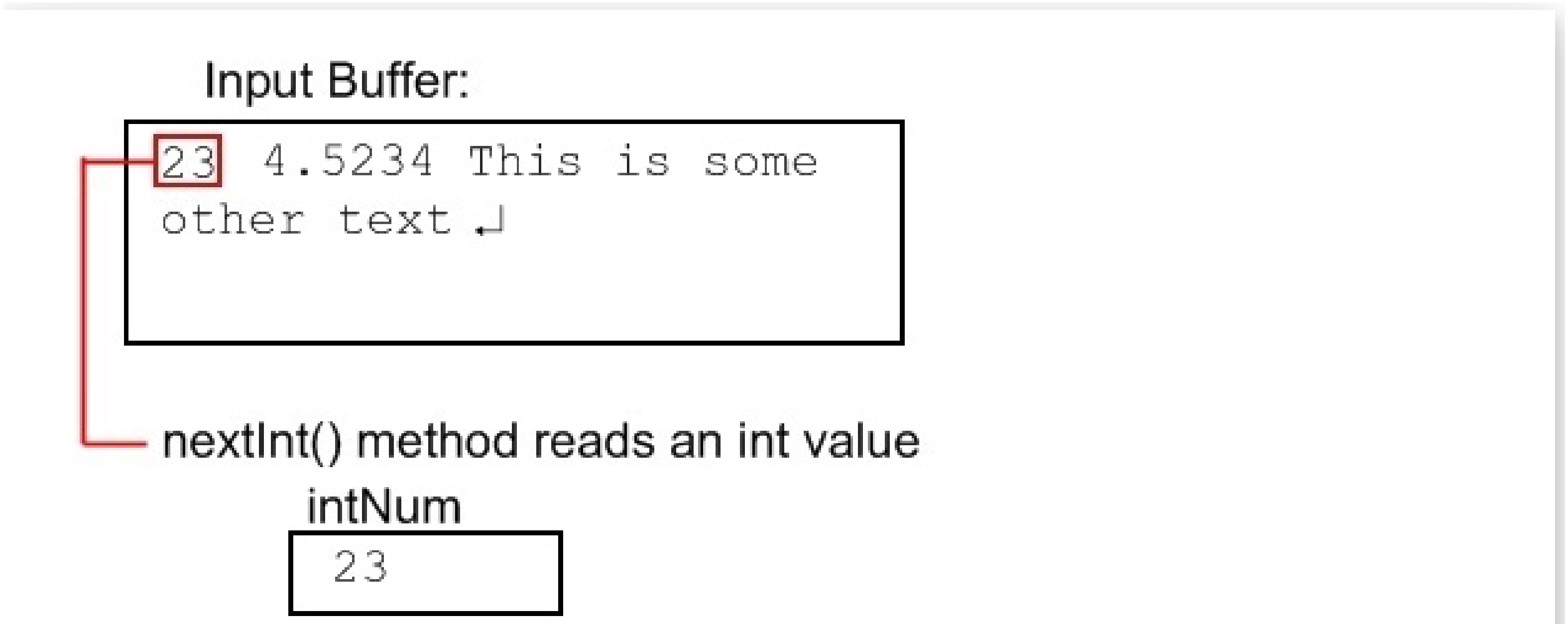
```
Enter an integer, then a double, then type some text.
23 4.5234 This is some other text
23
4.5234
 This is some other text
Press any key to continue . . .
```

- Once you enter the complete line of values, they are stored in the input buffer.

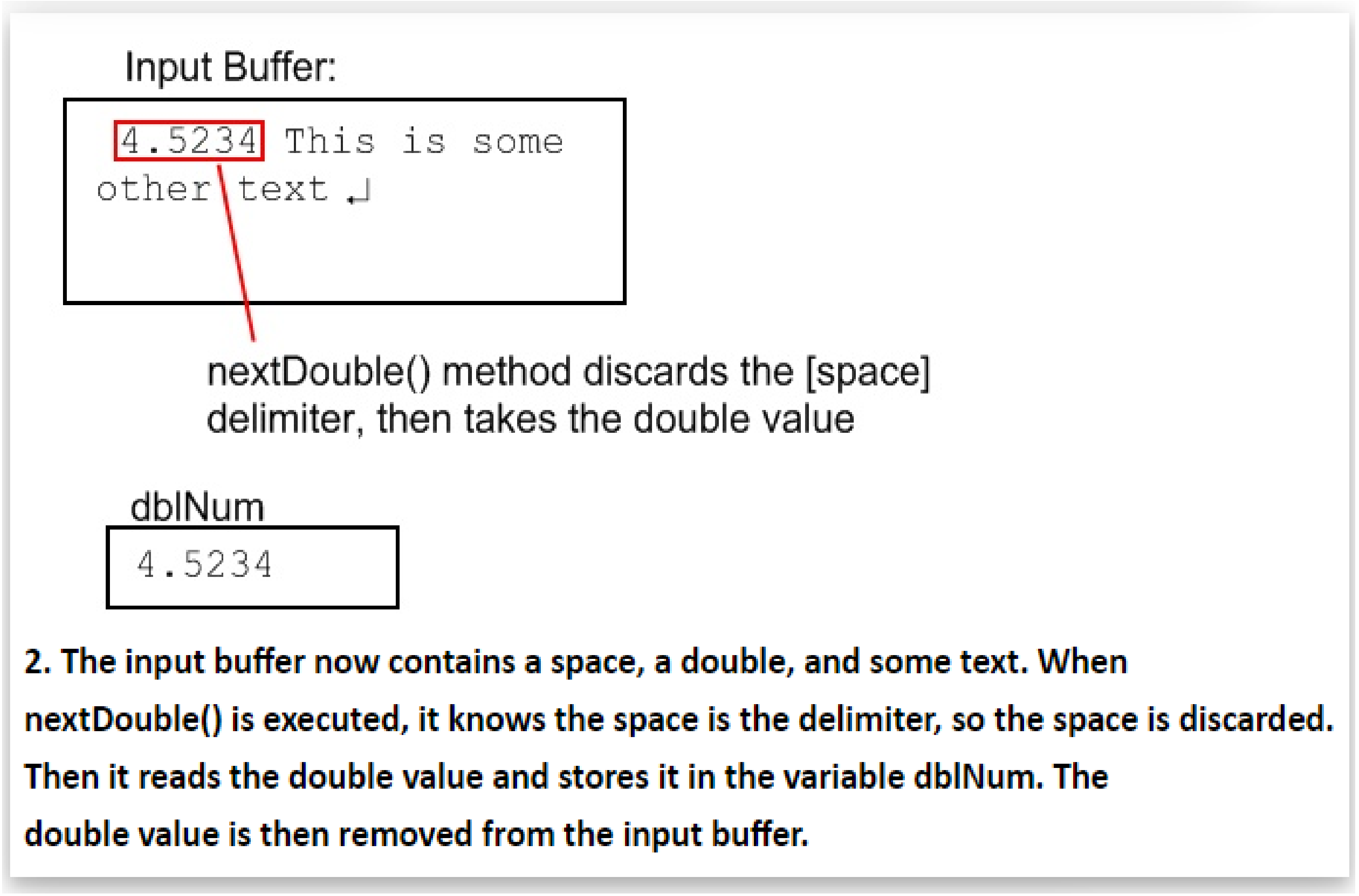
- When you call nextInt(), the Scanner object goes to the input buffer and reads back an integer value. That value (23) is then removed from the buffer, because it has been processed by the Scanner.

- When nextInt() is executing, it takes all the characters it finds until it encounters a white-space character (such as tab, space, or new-line).

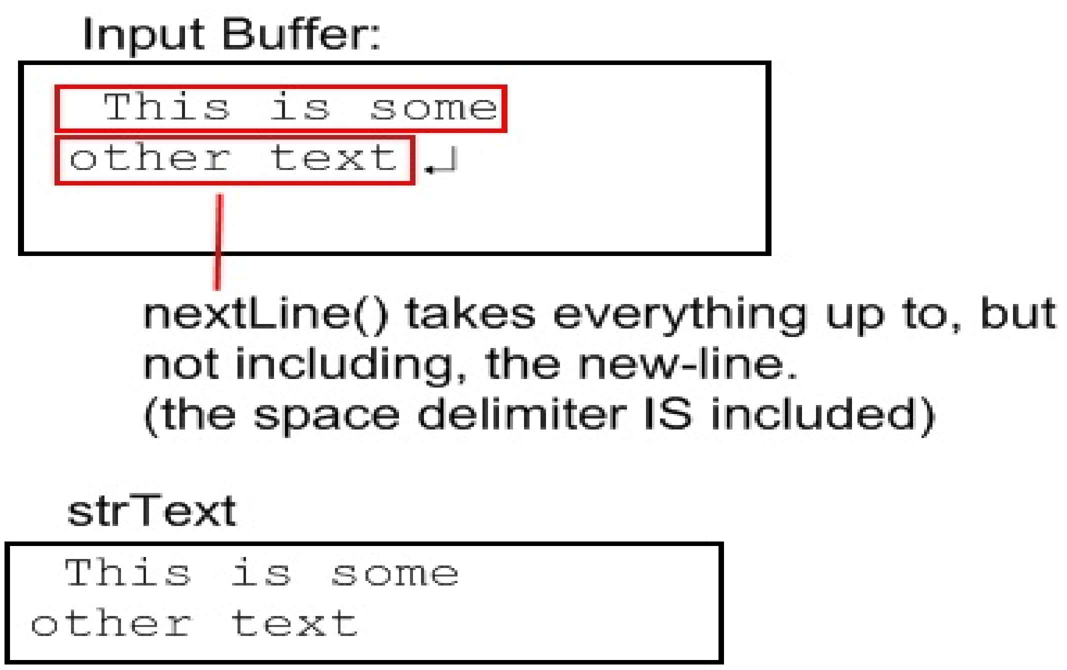
- Then it takes those characters and parses or converts them into an int value. It does not take the white-space character that indicates the end of the int value, so that space is still in the buffer after nextInt() executes.


Input Buffer:
23 4.5234 This is some other text ↵
nextInt() method reads an int value
intNum
23

1. When nextInt() is executed, it reads back the integer value and stores it in the intNum variable. Then the integer value is removed from the input buffer.

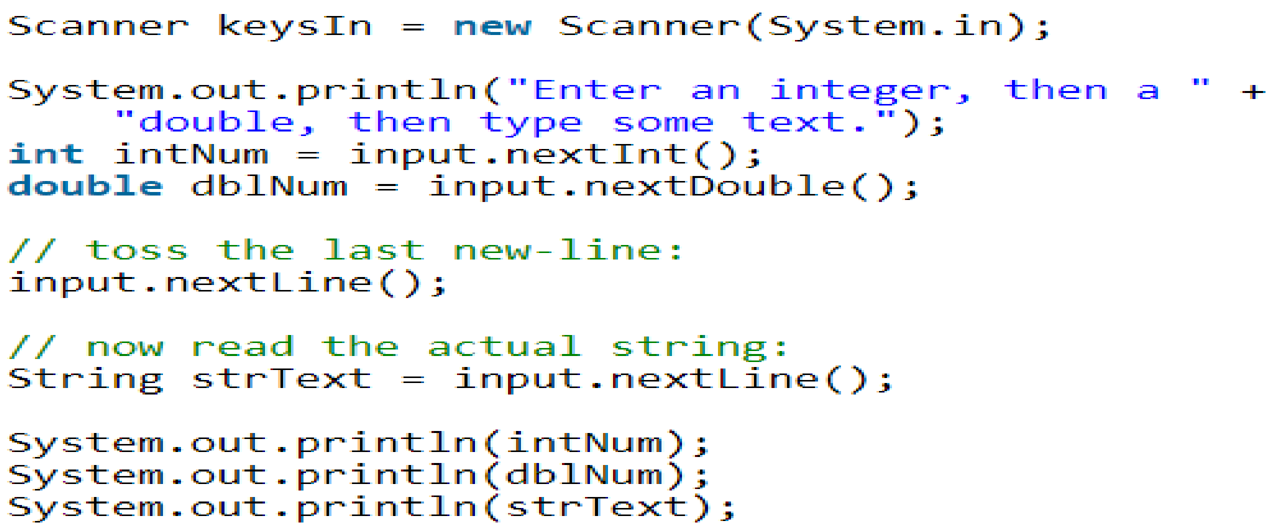

Input Buffer:
4.5234 This is some other text ↲
nextDouble() method discards the [space] delimiter, then takes the double value
dblNum
4.5234

2. The input buffer now contains a space, a double, and some text. When nextDouble() is executed, it knows the space is the delimiter, so the space is discarded. Then it reads the double value and stores it in the variable dblNum. The double value is then removed from the input buffer.



nextLine() takes everything up to, but not including, the new-line. (the space delimiter IS included)

3. Now the input buffer contains a space (the delimiter between the double and the text), and the rest of the text ("This is some other text"). The nextLine() method uses the new-line as the delimiter, so it removes all the text from the input buffer and stores it in the strText variable. It stops at the new-line. Then it takes the new-line out of the buffer and discards it.

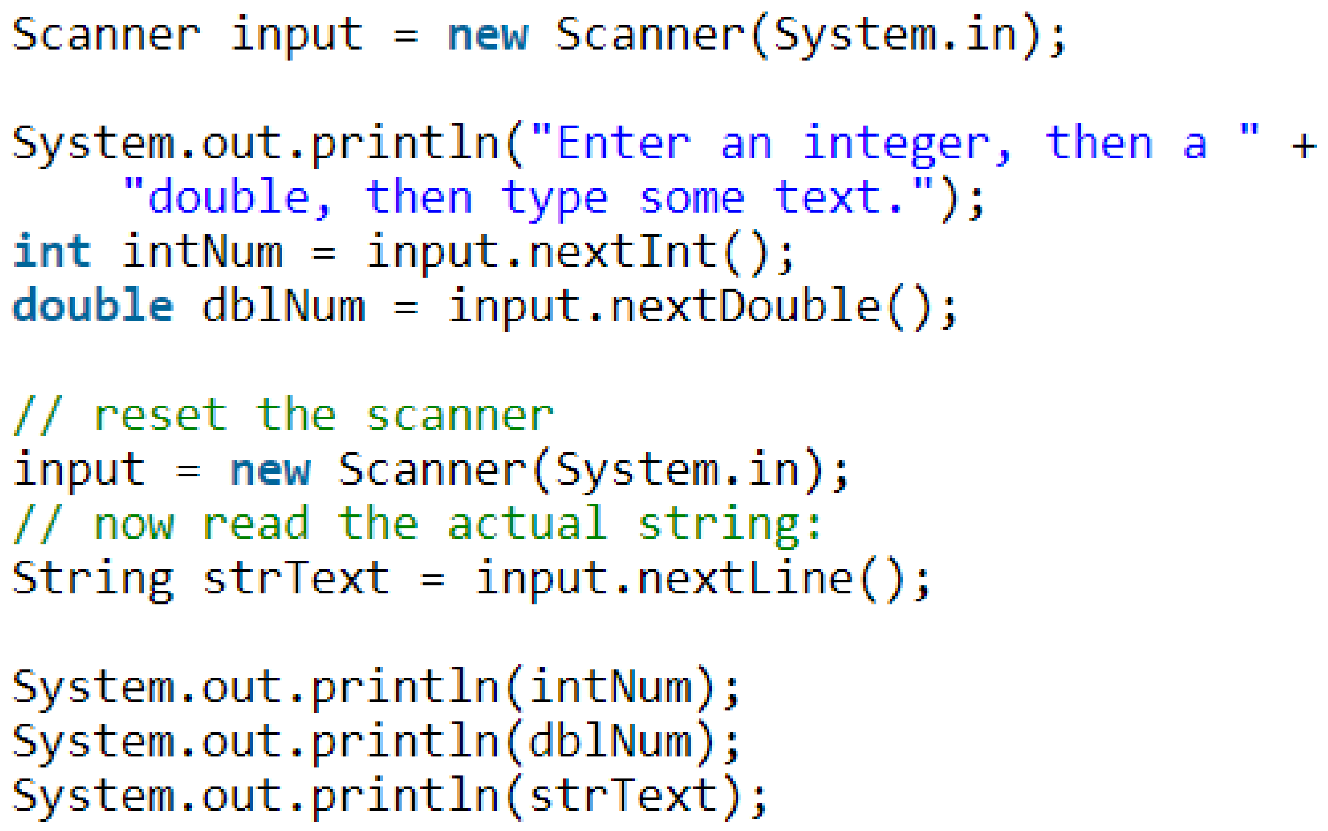


```
Scanner keysIn = new Scanner(System.in);

System.out.println("Enter an integer, then a " +
     "double, then type some text.");
int intNum = input.nextInt();
double dblNum = input.nextDouble();

// toss the last new-line:
input.nextLine();

// now read the actual string:
String strText = input.nextLine();

System.out.println(intNum);
System.out.println(dblNum);
System.out.println(strText);
```

When you run the modified program and press enter after each input, you'll get something like:

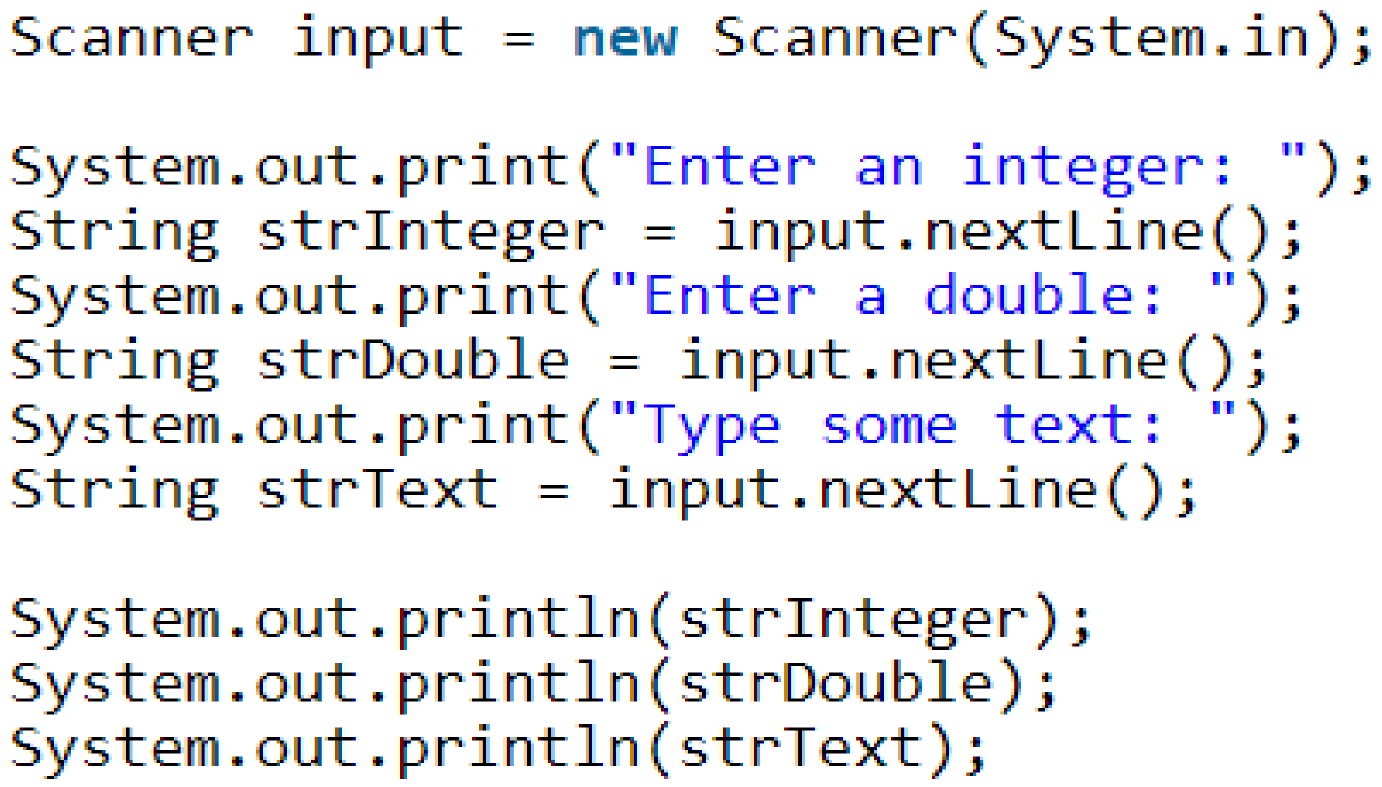
```
Enter an integer, then a double, then type some text.
23
2.5
This is a line of text.
23
2.5
This is a line of text.
```

When you run the program, you should get the same output as the previous example.



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

System.out.println("Enter an integer, then a " +
    "double, then type some text.");
int intNum = input.nextInt();
double dblNum = input.nextDouble();

// reset the scanner
input = new Scanner(System.in);
// now read the actual string:
String strText = input.nextLine();

System.out.println(intNum);
System.out.println(dblNum);
System.out.println(strText);
```

A second alternative is to always capture your input using the nextLine() method, then you never have to worry about extra characters in the input buffer. For example:



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

System.out.print("Enter an integer: ");
String strInteger = input.nextLine();
System.out.print("Enter a double: ");
String strDouble = input.nextLine();
System.out.print("Type some text: ");
String strText = input.nextLine();

System.out.println(strInteger);
System.out.println(strDouble);
System.out.println(strText);
```

Parse Methods

In Java, there are some special classes called **wrapper classes** that are associated with each primitive type.

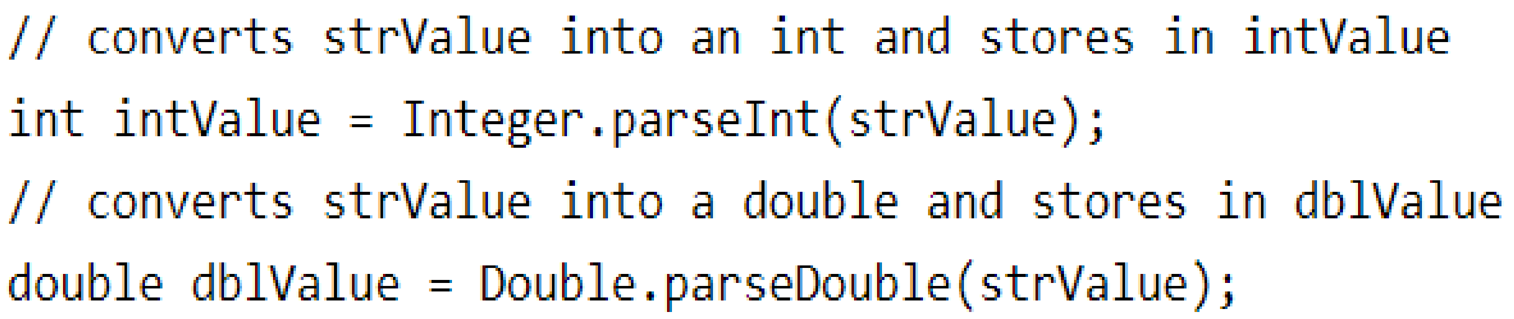
For example, the wrapper class for the int primitive type is called Integer, and the wrapper class for the double primitive type is called Double.

These wrapper classes contain methods that you can use to convert Strings into primitive types . For example:

```
String strIntNum = ”2”;
String strDblNum = ”2.5”;
int intNum = Integer.parselnt(strlntNum);
// stores 2 in intNum
    double dblNum = Double.parseDouble(strDblNum);

// stores 2.5 in dblNum
```

You will use the parse methods whenever you have a string value and you want to convert it into an int or a double.

Parse Methods



```
// converts strValue into an int and stores in intValue
int intValue = Integer.parseInt(strValue);
// converts strValue into a double and stores in dblValue
double dblValue = Double.parseDouble(strValue);
```

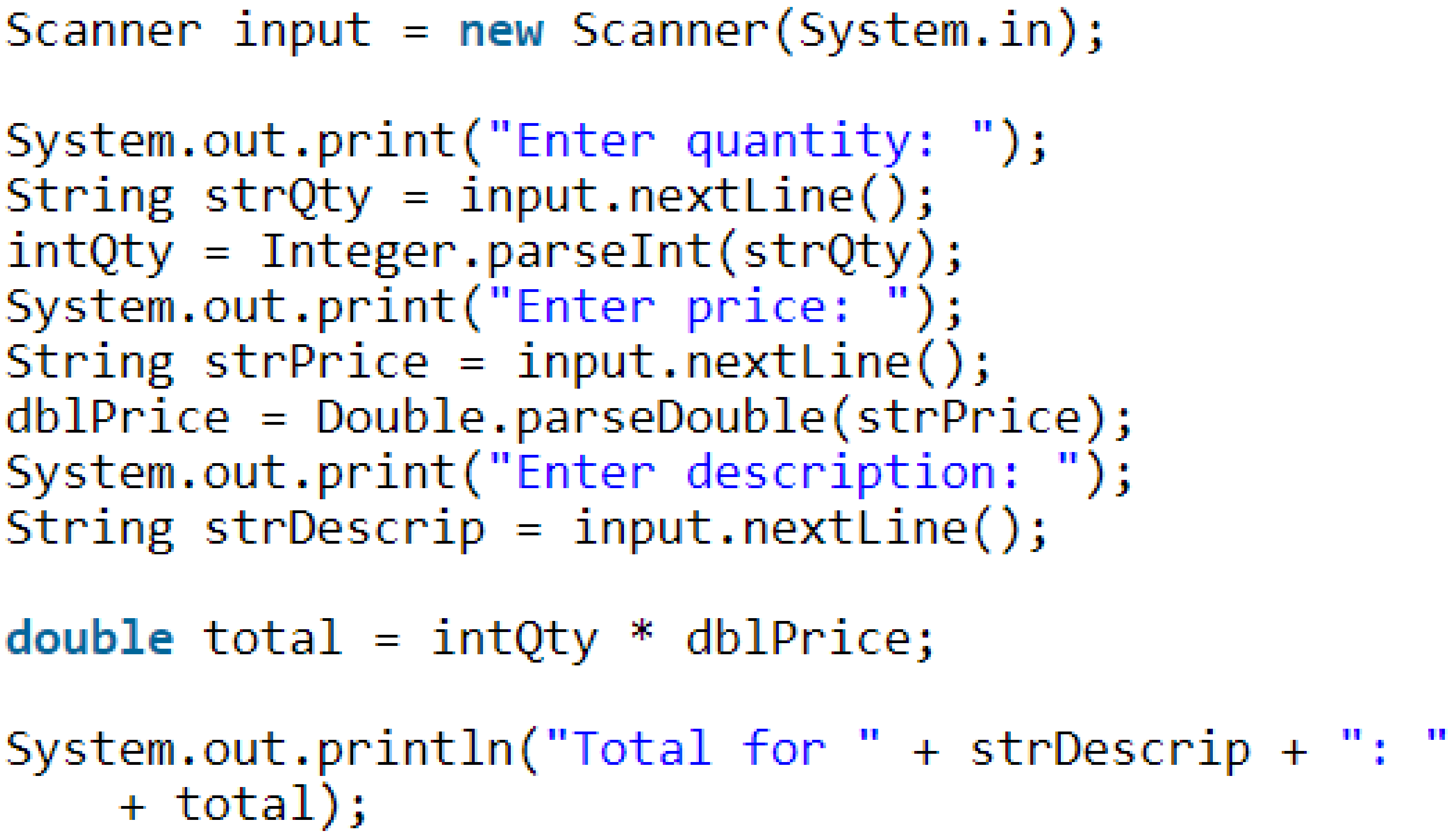
In the first statement, we call on the parseInt() method in the Integer class and give it strValue as an argument/input.

We can assume here that strValue is a String variable. The Integer.parseInt() method then converts strValue into a primitive int value and stores that result into the variable intValue.

In the second statement, we call on the parseDouble() method in the Double class and give it strValue as an argument/input.

Again, this example assumes strValue is a String variable. The Double.parseDouble() method converts strValue into a primitive double and stores that result into the variable dblValue.

You can use either of these methods to convert user-input values into the appropriate data type. For example:



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

System.out.print("Enter quantity: ");
String strQty = input.nextLine();
intQty = Integer.parseInt(strQty);
System.out.print("Enter price: ");
String strPrice = input.nextLine();
dblPrice = Double.parseDouble(strPrice);
System.out.print("Enter description: ");
String strDescrip = input.nextLine();

double total = intQty * dblPrice;

System.out.println("Total for " + strDescrip + ": "
    + total);
```

One thing to note about using Integer.parseInt() and Double.parseDouble() is that they will crash if they attempt to parse anything that isn't a valid int or double.

For integers, this means anything that isn't a digit (so any value with letters, symbols including the decimal point and comma, or even the null-string ("") will cause the program to crash).

For double values this means only digits and one decimal point are allowed; any commas, letters, symbols, multiple decimal points, and the null-string will cause the program to crash.

In a later lesson, you'll learn how to validate the user's input before you parse it, so that the program won't crash! In fact, you'll even be able to give the user a nice error message and even prompt them to try entering the value again. :)

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();
```

ComputeAreaWithConsoleInput

Run

ComputeAverage

Run

Reading Input from the Console

1. Create a Scanner object

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

2. Use the method nextDouble() to obtain to a double value. For example,

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

https://liveexample.pearsoncmg.com//html/ComputeAreaWithConsoleInput.html

ComputeAreaWithConsoleInput Run

ComputeAverage Run

https://liveexample.pearsoncmg.com//html/ComputeAverage.html

Example

```
import java.util.Scanner;
public class Exercise02_01Extra {
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter the width and height of a rectangle: ");
    double w = input.nextDouble();
    double h = input.nextDouble();
    double perimeter = 2 * (w + h);
    double area = w * h;
    System.out.println("The perimeter is " + perimeter);
    System.out.println("The area is " + area);

  }
}
```

Displaying Text in a Message Dialog Box

you can use the showMessageDialog method in the JOptionPane class. JOptionPane is one of the many predefined classes in the Java system, which can be reused rather than "reinventing the wheel."

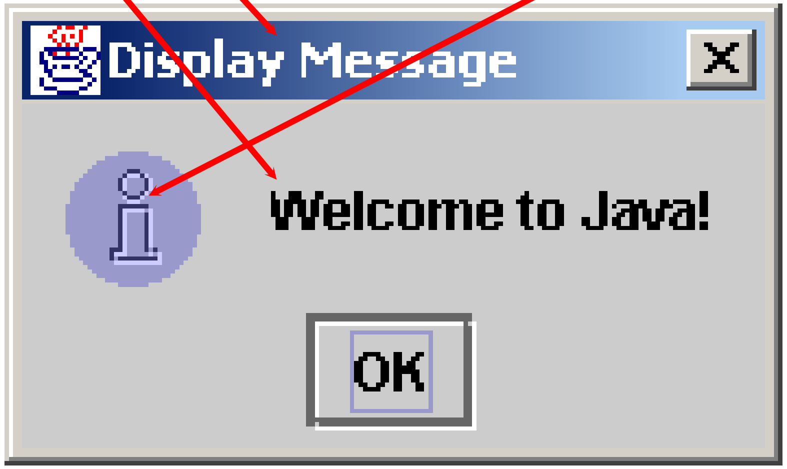
WelcomeInMessageDialogBox

Run

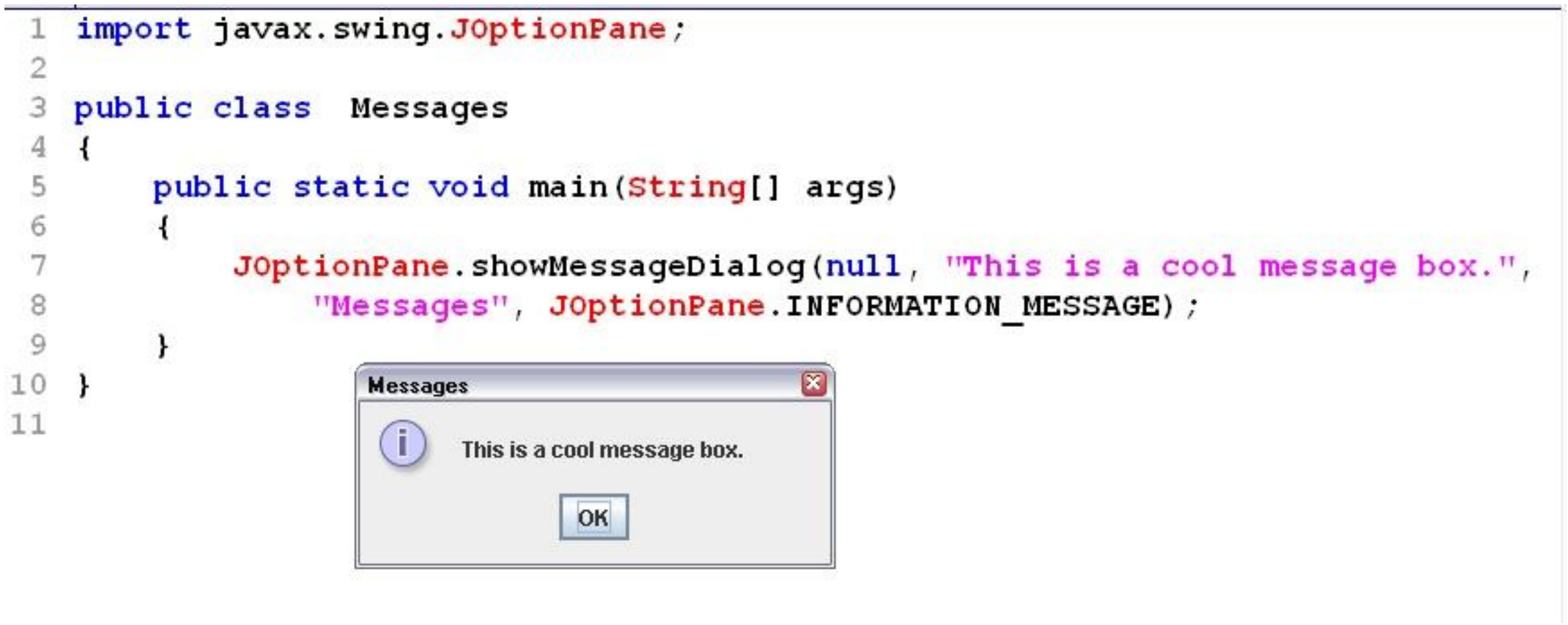
IMPORTANT NOTE: (1) To enable the buttons, you must download the entire slide file *slide.zip* and unzip the files into a directory (e.g., c:\slide) . (2) You must have installed JDK and set JDK's bin directory in your environment path (e.g., c:\Program Files\java\jdk1.7.0\bin in your environment path. (3) If you are using Office 2010, check PowerPoint2010.doc located in the same folder with this ppt file.

The showMessageDialog Method

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



JOptionPane Code



```java
import javax.swing.JOptionPane;

public class  Messages
{
    public static void main(String[] args)
    {
        JOptionPane.showMessageDialog(null, "This is a cool message box.",
            "Messages", JOptionPane.INFORMATION_MESSAGE);
    }
}

```

Messages

This is a cool message box.

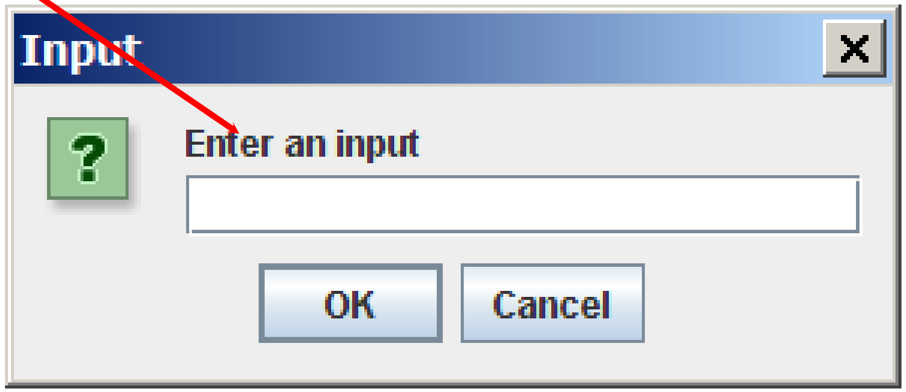
JOptionPane Input

This book provides two ways of obtaining input.

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

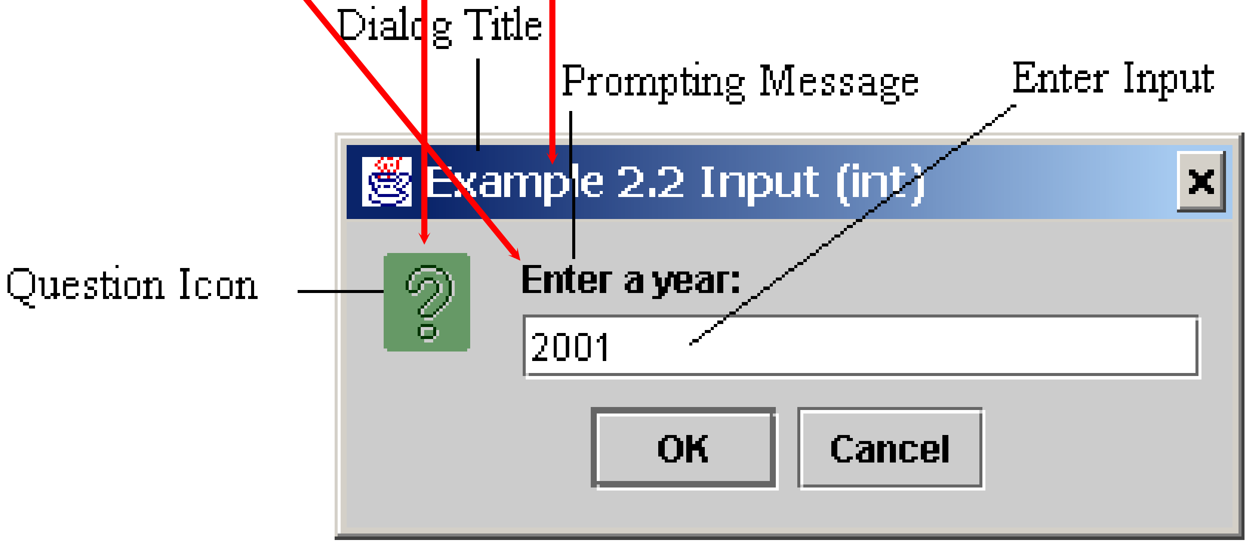
Getting Input from Input Dialog Boxes

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, x,
  y, JOptionPane.QUESTION_MESSAGE);
```

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

The other is to use a statement like this:

```
JOptionPane.showInputDialog(x);
```

where x is a string for the prompting message.