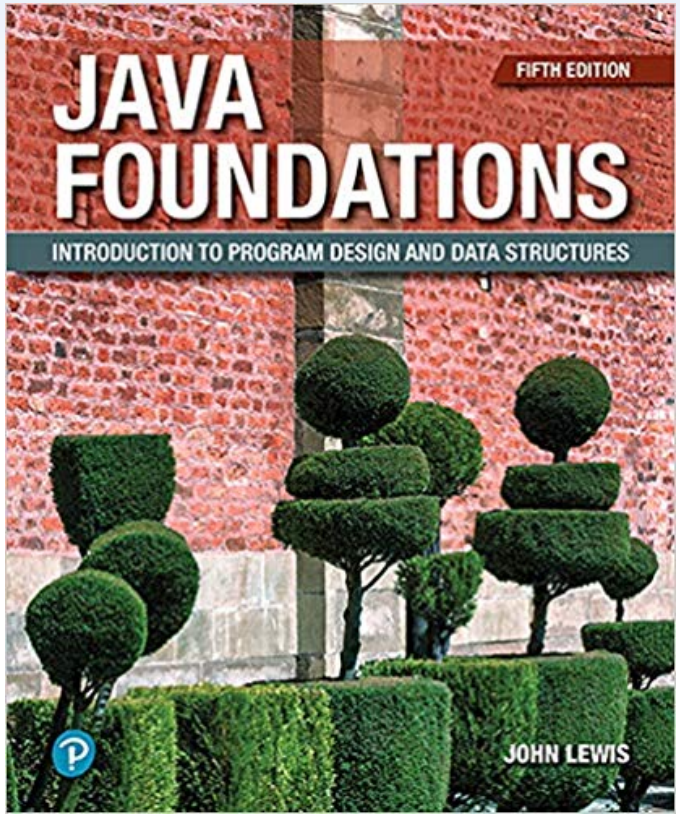


Chapter 2

Data and Expressions

Chapter Scope

- Character strings and concatenation
- Escape sequences
- Declaring and using variables
- Java primitive types
- Expressions
- Data conversions
- The `Scanner` class for interactive programs

Character Strings

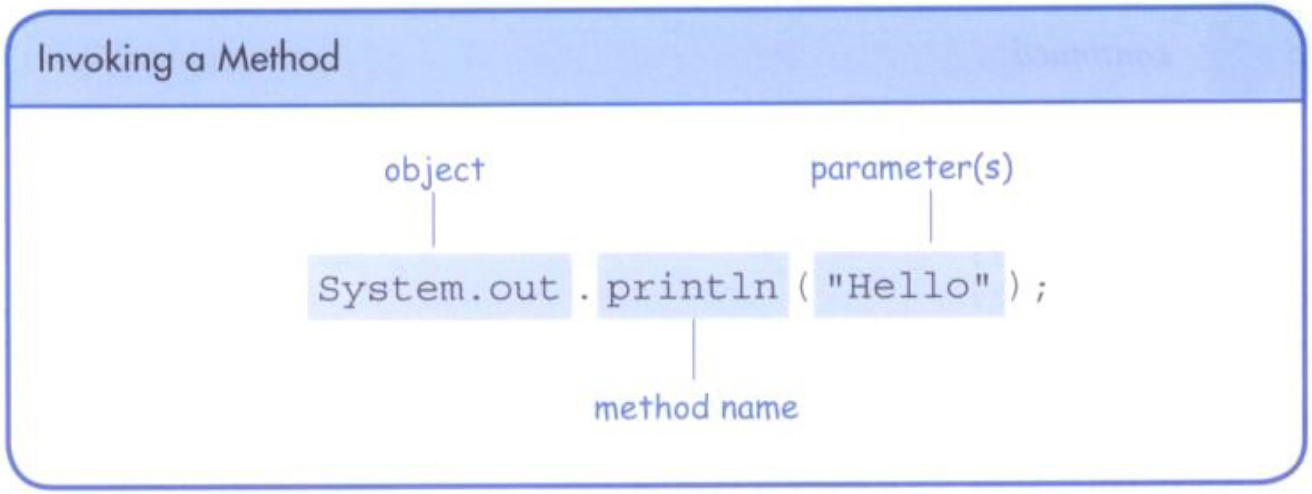
- A string of characters can be represented as a *string literal* by putting double quotes around it
- Examples:

```
"This is a string literal."
"123 Main Street"
"X"
```

- Every character string is an object in Java, defined by the `String` class
- Every string literal represents a `String` object

The println Method

- In the `Lincoln` program, we invoked the `println` method to print a character string
- The `System.out` object represents a destination (the monitor) to which we can send output



The print Method

- The `System.out` object provides another service as well
- The `print` method is similar to the `println` method, except that it does not advance to the next line
- Therefore anything printed after a `print` statement will appear on the same line

```
//********************************************************************
//  Countdown.java       Java Foundations
//
//  Demonstrates the difference between print and println.
//********************************************************************

public class Countdown
{
   //-----------------------------------------------------------------
   //  Prints two lines of output representing a rocket countdown.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.print("Three... ");
      System.out.print("Two... ");
      System.out.print("One... ");
      System.out.print("Zero... ");

      System.out.println("Liftoff!"); // appears on first output line

      System.out.println("Houston, we have a problem.");
   }
}
```

String Concatenation

- The *string concatenation operator* (+) is used to append one string to the end of another

```
"Peanut butter " + "and jelly"
```

- It can also be used to append a number to a string
- A string literal cannot be broken across two lines in a program

```
//********************************************************************
//  Facts.java       Java Foundations
//
//  Demonstrates the use of the string concatenation operator and the
//  automatic conversion of an integer to a string.
//********************************************************************

public class Facts
{
   //-----------------------------------------------------------------
   //  Prints various facts.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      // Strings can be concatenated into one long string
      System.out.println("We present the following facts for your "
                         + "extracurricular edification:");

      System.out.println();

      // A string can contain numeric digits
      System.out.println("Letters in the Hawaiian alphabet: 12");

      // A numeric value can be concatenated to a string
      System.out.println("Dialing code for Antarctica: " + 672);

      System.out.println("Year in which Leonardo da Vinci invented "
                         + "the parachute: " + 1515);

      System.out.println("Speed of ketchup: " + 40 + " km per year");
   }
}
```

String Concatenation

- The + operator is also used for arithmetic addition
- The function that it performs depends on the type of the information on which it operates
- If both operands are strings, or if one is a string and one is a number, it performs string concatenation
- If both operands are numeric, it adds them
- The + operator is evaluated left to right, but parentheses can be used to force the order

```
//********************************************************************
//  Addition.java       Java Foundations
//
//  Demonstrates the difference between the addition and string
//  concatenation operators.
//********************************************************************

public class Addition
{
   //-----------------------------------------------------------------
   //  Concatenates and adds two numbers and prints the results.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.println("24 and 45 concatenated: " + 24 + 45);

      System.out.println("24 and 45 added: " + (24 + 45));
   }
}
```

Escape Sequences

- What if we wanted to print a the quote character?
- The following line would confuse the compiler because it would interpret the second quote as the end of the string

```
System.out.println("I said "Hello" to you.");
```

- An *escape sequence* is a series of characters that represents a special character
- An escape sequence begins with a backslash character (\)

```
System.out.println("I said \"Hello\" to you.");
```

Escape Sequences

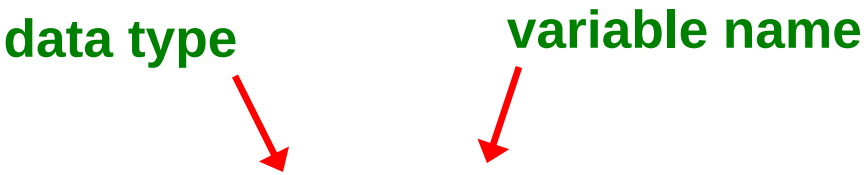
- Some Java escape sequences:

Escape Sequence	Meaning
\b	backspace
\t	tab
\n	newline
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

```
//********************************************************************
//  Roses.java       Java Foundations
//
//  Demonstrates the use of escape sequences.
//********************************************************************

public class Roses
{
   //-----------------------------------------------------------------
   //  Prints a poem (of sorts) on multiple lines.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      System.out.println("Roses are red,\n\tViolets are blue,\n" +
         "Sugar is sweet,\n\tBut I have \"commitment issues\",\n\t" +
         "So I'd rather just be friends\n\tAt this point in our " +
         "relationship.");
   }
}
```

Variables

- A *variable* is a name for a location in memory
- A variable must be *declared* by specifying its name and the type of information that it will hold

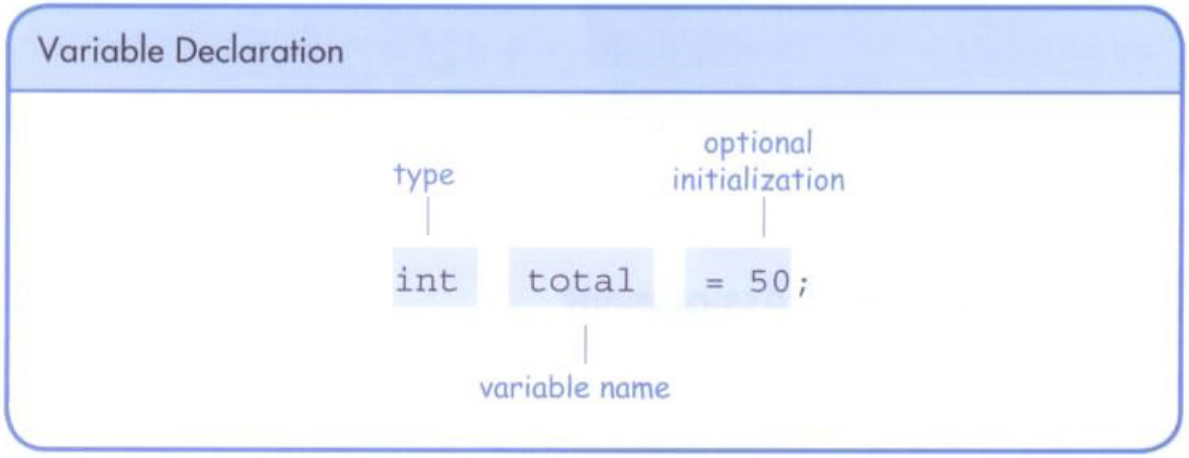



```
int total;

int count, temp, result;
```

Multiple variables can be created in one declaration

Variables

- A variable can be given an initial value in the declaration



- When a variable is used in a program, its current value is used

```
//********************************************************************
//  PianoKeys.java       Java Foundations
//
//  Demonstrates the declaration, initialization, and use of an
//  integer variable.
//********************************************************************

public class PianoKeys
{
   //-----------------------------------------------------------------
   //  Prints the number of keys on a piano.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int keys = 88;

      System.out.println("A piano has " + keys + " keys.");
   }
}
```

Assignment

- An *assignment statement* changes the value of a variable
- The assignment operator is the = sign

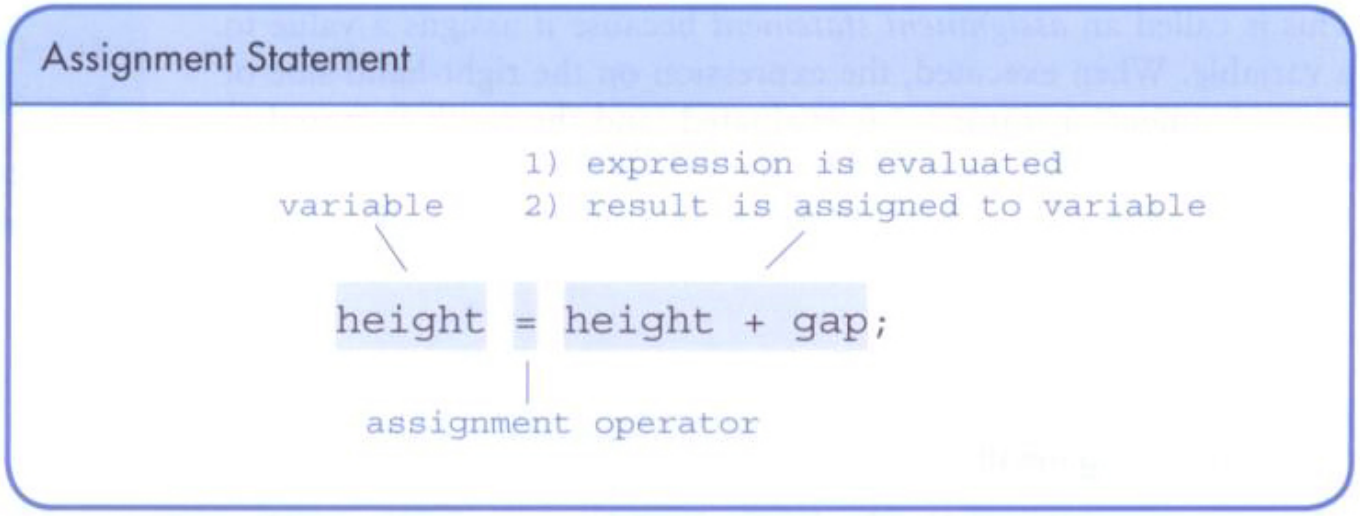
```
total = 55;
```

- The expression on the right is evaluated and the result is stored in the variable on the left
- The value that was in `total` is overwritten
- You can only assign a value to a variable that is consistent with the variable's declared type

```
//********************************************************************
//  Geometry.java       Java Foundations
//
//  Demonstrates the use of an assignment statement to change the
//  value stored in a variable.
//********************************************************************

public class Geometry
{
   //-----------------------------------------------------------------
   //  Prints the number of sides of several geometric shapes.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int sides = 7;  // declaration with initialization
      System.out.println("A heptagon has " + sides + " sides.");

      sides = 10;  // assignment statement
      System.out.println("A decagon has " + sides + " sides.");

      sides = 12;
      System.out.println("A dodecagon has " + sides + " sides.");
   }
}
```

Assignment

- The right-hand side could be an expression
- The expression is completely evaluated and the result is stored in the variable



Constants

- A *constant* is an identifier that is similar to a variable except that it holds the same value during its entire existence
- As the name implies, it is constant, not variable
- The compiler will issue an error if you try to change the value of a constant
- In Java, we use the `final` modifier to declare a constant

```
final int MIN_HEIGHT = 69;
```

Constants

- Constants are useful for three important reasons
  - First, they give meaning to otherwise unclear literal values
    - For example, MAX_LOAD means more than the literal 250
  - Second, they facilitate program maintenance
    - If a constant is used in multiple places, its value need only be updated in one place
  - Third, they formally establish that a value should not change, avoiding inadvertent errors by other programmers

Primitive Data Types

- There are eight primitive data types in Java
- Four of them represent integers
  - `byte`, `short`, `int`, `long`
- Two of them represent floating point numbers
  - `float`, `double`
- One of them represents characters
  - `char`
- And one of them represents boolean values
  - `boolean`

Numeric Types

- The difference between the various numeric primitive types is their size, and therefore the values they can store:

Type	Storage	Min Value	Max Value
byte	8 bits	–128	127
short	16 bits	–32,768	32,767
int	32 bits	–2,147,483,648	2,147,483,647
long	64 bits	–9,223,372,036,854,775,808	9,223,372,036,854,775,807
float	32 bits	Approximately –3.4E+38 with 7 significant digits	Approximately 3.4E+38 with 7 significant digits
double	64 bits	Approximately –1.7E+308 with 15 significant digits	Approximately 1.7E+308 with 15 significant digits

Characters

- A `char` variable stores a single character
- Character literals are delimited by single quotes:

```
'a'   'X'   '7'   '$'   ','   '\n'
```

- Example declarations

```
char topGrade = 'A';

char terminator = ';', separator = ' ';
```

- Note the distinction between a primitive character variable, which holds only one character, and a `String` object, which can hold multiple characters

Character Sets

- A *character set* is an ordered list of characters, with each character corresponding to a unique number
- A `char` variable in Java can store any character from the *Unicode character set*
- The Unicode character set uses sixteen bits per character
- It is an international character set, containing symbols and characters from many world languages

Characters

- The *ASCII character set* is older and smaller than Unicode
- The ASCII characters are a subset of the Unicode character set, including:

uppercase letters	**A, B, C, …**
lowercase letters	**a, b, c, …**
punctuation	**period, semi-colon, …**
digits	**0, 1, 2, …**
special symbols	**&, |, \, …**
control characters	**carriage return, tab, ...**

Booleans

- A `boolean` value represents a true or false condition
- The reserved words `true` and `false` are the only valid values for a boolean type

```
boolean done = false;
```

- A `boolean` variable can also be used to represent any two states, such as a light bulb being on or off

Expressions

- An *expression* is a combination of one or more operators and operands
- *Arithmetic expressions* compute numeric results and make use of the arithmetic operators
  - Addition +
  - Subtraction -
  - Multiplication *
  - Division /
  - Remainder %
- If either or both operands used by an arithmetic operator are floating point, then the result is a floating point

Division and Remainder

- If both operands to the division operator (/) are integers, the result is an integer (the fractional part is discarded)

```
14 / 3      equals      4

8 / 12      equals      0
```

- The remainder operator (%) returns the remainder after dividing the second operand into the first

```
14 % 3      equals      2

8 % 12      equals      8
```

Operator Precedence

- Operators can be combined into complex expressions

```
result = total + count / max - offset;
```

- Operators have a well-defined precedence which determines the order in which they are evaluated
- Multiplication, division, and remainder are evaluated prior to addition, subtraction, and string concatenation
- Arithmetic operators with the same precedence are evaluated from left to right, but parentheses can be used to force the evaluation order

Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
```

```
a + b * c - d / e
```

```
a / (b + c) - d % e
```

```
a / (b * (c + (d - e)))
```

Operator Precedence

- What is the order of evaluation in the following expressions?

```
a + b + c + d + e
  1   2   3   4
```

```
a + b * c - d / e
  3   1   4   2
```

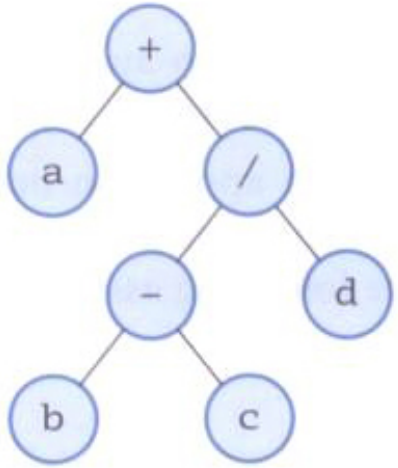
```
a / (b + c) - d % e
  2    1     4   3
```

```
a / (b * (c + (d - e)))
  4    3    2    1
```

Expression Trees

- The evaluation of a particular expression can be shown using an *expression tree*
- The operators lower in the tree have higher precedence for that expression

Evaluating
a + (b – c) / d



Operator Precedence

- Precedence among some Java operators:

Precedence Level	Operator	Operation	Associates
1	+	unary plus	R to L
	–	unary minus	
2	*	multiplication	L to R
	/	division	
	%	remainder	
3	+	addition	L to R
	–	subtraction	
	+	string concatenation	
4	=	assignment	R to L

```
//********************************************************************
//  TempConverter.java       Java Foundations
//
//  Demonstrates the use of primitive data types and arithmetic
//  expressions.
//********************************************************************

public class TempConverter
{
   //-----------------------------------------------------------------
   //  Computes the Fahrenheit equivalent of a specific Celsius
   //  value using the formula F = (9/5)C + 32.
   //-----------------------------------------------------------------
   public static void main (String[] args)
   {
      final int BASE = 32;
      final double CONVERSION_FACTOR = 9.0 / 5.0;

      double fahrenheitTemp;
      int celsiusTemp = 24;  // value to convert

      fahrenheitTemp = celsiusTemp * CONVERSION_FACTOR + BASE;

      System.out.println ("Celsius Temperature: " + celsiusTemp);
      System.out.println ("Fahrenheit Equivalent: " + fahrenheitTemp);
   }
}
```

Assignment Revisited

- The assignment operator has a lower precedence than the arithmetic operators

First the expression on the right hand side of the = operator is evaluated

```
answer = sum / 4 + MAX * lowest;
         4     1   3     2
```

Then the result is stored in the variable on the left hand side

Assignment Revisited

- The right and left hand sides of an assignment statement can contain the same variable

First, one is added to the original value of count

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

Then the result is stored back into count (overwriting the original value)

Increment and Decrement Operators

- The increment and decrement operators use only one operand
- The *increment operator* (++) adds one to its operand
- The *decrement operator* (--) subtracts one from its operand
- The statement

```
count++;
```

  is functionally equivalent to

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

Increment and Decrement Operators

- The increment and decrement operators can be applied in *postfix form*

```
count++
```

- or *prefix form*

```
++count
```

- When used as part of a larger expression, the two forms can have different effects
- Because of their subtleties, the increment and decrement operators should be used with care

Assignment Operators

- Often we perform an operation on a variable, and then store the result back into that variable
- Java provides *assignment operators* to simplify that process
- For example, the statement

```
num += count;
```

is equivalent to

```
num = num + count;
```

Assignment Operators

- There are many assignment operators in Java, including the following:

Operator	Example	Equivalent To
`+=`	`x += y`	`x = x + y`
`-=`	`x -= y`	`x = x - y`
`*=`	`x *= y`	`x = x * y`
`/=`	`x /= y`	`x = x / y`
`%=`	`x %= y`	`x = x % y`

Assignment Operators

- The right hand side of an assignment operator can be a complex expression
- The entire right-hand expression is evaluated first, then the result is combined with the original variable
- Therefore

```
result /= (total-MIN) % num;
```

is equivalent to

```
result = result / ((total-MIN) % num);
```

Assignment Operators

- The behavior of some assignment operators depends on the types of the operands
- If the operands to the += operator are strings, the assignment operator performs string concatenation
- The behavior of an assignment operator (+=) is always consistent with the behavior of the corresponding operator (+)

Data Conversions

- Sometimes it is convenient to convert data from one type to another
- For example, in a particular situation we may want to treat an integer as a floating point value
- These conversions do not change the type of a variable or the value that's stored in it – they only convert a value as part of a computation

Data Conversions

- Conversions must be handled carefully to avoid losing information
- *Widening conversions* are safest because they tend to go from a small data type to a larger one (such as a `short` to an `int`)
- *Narrowing conversions* can lose information because they tend to go from a large data type to a smaller one.
- In Java, data conversions can occur in three ways
  - assignment conversion
  - promotion
  - casting

Data Conversions

Widening Conversions

From	To
byte	short, int, long, float, or double
short	int, long, float, or double
char	int, long, float, or double
int	long, float, or double
long	float or double
float	double

Narrowing Conversions

From	To
byte	char
short	byte or char
char	byte or short
int	byte, short, or char
long	byte, short, char, or int
float	byte, short, char, int, or long
double	byte, short, char, int, long, or float

Assignment Conversion

- *Assignment conversion* occurs when a value of one type is assigned to a variable of another
- If `money` is a `float` variable and `dollars` is an `int` variable, the following assignment converts the value in `dollars` to a `float`

```
money = dollars
```

- Only widening conversions can happen via assignment
- Note that the value or type of `dollars` did not change

Promotion

- *Promotion* happens automatically when operators in expressions convert their operands
- For example, if `sum` is a `float` and `count` is an `int`, the value of `count` is converted to a floating point value to perform the following calculation

```
result = sum / count;
```

Casting

- *Casting* is the most powerful, and dangerous, technique for conversion
- Both widening and narrowing conversions can be accomplished by explicitly casting a value
- To cast, the type is put in parentheses in front of the value being converted
- For example, if `total` and `count` are integers, but we want a floating point result when dividing them, we can cast `total`

```
result = (float) total / count;
```

The Scanner Class

- The `Scanner` class provides convenient methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources, including the user typing values on the keyboard
- Keyboard input is represented by the `System.in` object

Reading Input

- The following line creates a `Scanner` object that reads from the keyboard

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

- The `new` operator creates the `Scanner` object
- Once created, the `Scanner` object can be used to invoke various input methods, such as

```
answer = scan.nextLine();
```

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
- The `nextLine` method reads all of the input until the end of the line is found
- We'll discuss the details of object creation and class libraries later

- Some methods of the Scanner class:

```
Scanner (InputStream source)
Scanner (File source)
Scanner (String source)
```
Constructors: sets up the new scanner to scan values from the specified source.

```
String next()
```
Returns the next input token as a character string.

```
String nextLine()
```
Returns all input remaining on the current line as a character string.

```
boolean nextBoolean()
byte nextByte()
double nextDouble()
float nextFloat()
int nextInt()
long nextLong()
short nextShort()
```
Returns the next input token as the indicated type. Throws `InputMismatchException` if the next token is inconsistent with the type.

```
boolean hasNext()
```
Returns true if the scanner has another token in its input.

```
Scanner useDelimiter (String pattern)
Scanner useDelimiter (Pattern pattern)
```
Sets the scanner's delimiting pattern.

```
Pattern delimiter()
```
Returns the pattern the scanner is currently using to match delimiters.

```
String findInLine (String pattern)
String findInLine (Pattern pattern)
```
Attempts to find the next occurrence of the specified pattern, ignoring delimiters.

```
//********************************************************************
//  Echo.java       Java Foundations
//
//  Demonstrates the use of the nextLine method of the Scanner class
//  to read a string from the user.
//********************************************************************

import java.util.Scanner;

public class Echo
{
   //-----------------------------------------------------------------
   //  Reads a character string from the user and prints it.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      String message;
      Scanner scan = new Scanner(System.in);

      System.out.println("Enter a line of text:");

      message = scan.nextLine();

      System.out.println("You entered: \"" + message + "\"");
   }
}
```

Input Tokens

- Unless specified otherwise, *white space* is used to separate the elements (called *tokens*) of the input
- White space includes space characters, tabs, new line characters
- The `next` method of the `Scanner` class reads the next input token and returns it as a string
- Methods such as `nextInt` and `nextDouble` read data of particular types

```
//********************************************************************
//  GasMileage.java       Java Foundations
//
//  Demonstrates the use of the Scanner class to read numeric data.
//********************************************************************

import java.util.Scanner;

public class GasMileage
{
   //-----------------------------------------------------------------
   //  Calculates fuel efficiency based on values entered by the
   //  user.
   //-----------------------------------------------------------------
   public static void main(String[] args)
   {
      int miles;
      double gallons, mpg;

      Scanner scan = new Scanner(System.in);

      System.out.print("Enter the number of miles: ");
      miles = scan.nextInt();

      System.out.print("Enter the gallons of fuel used: ");
      gallons = scan.nextDouble();

      mpg = miles / gallons;

      System.out.println("Miles Per Gallon: " + mpg);
   }
}
```