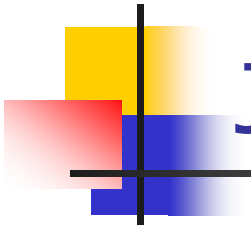
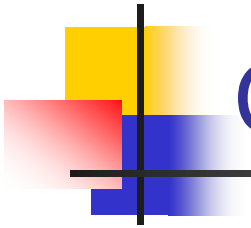


Java Basics

- First Java Program
- Comments
- Class Name / Source Code Filename
- `main` Method Heading
- Braces
- `System.out.println`
- Compilation and Execution
- Program Template
- Identifiers
- Variables
- Assignment Statements
- Initialization Statements

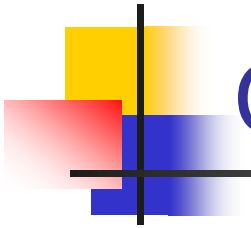


Chapter 3 - Java Basics

- Numeric Data Types – `int`, `long`
- Numeric Data Types – `float`, `double`
- Constants
- Arithmetic Operators
- Expression Evaluation
- Increment and Decrement Operators
- Compound Assignment Operators
- Type Casting
- Character Type - `char`
- Escape Sequences
- Primitive Variables vs. Reference Variables
- `String` Basics
- `String` Methods:
  - `equals`, `equalsIgnoreCase`, `length`, `charAt`
- Input - the `Scanner` Class

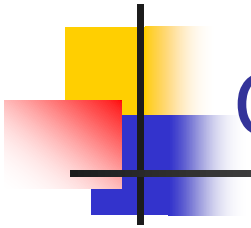
First Java Program

```
/**************************************
* Hello.java
* John Dean
*
* This program prints a hello message.
**************************************/

public class Hello
{
  public static void main(String[] args)
  {
    System.out.println("Hello, world!");
  }
} // end class Hello
```



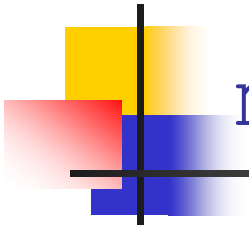
Comments

- Include comments in your programs in order to make them more readable/understandable.
- Block comment syntax:
  `/* ... */` (Note: The /* and */ can optionally span multiple lines)
- One line comment syntax:
  // ...
- Commented text is ignored by the compiler.
- Style requirement: Include a prologue section at the top of every program. The prologue section consists of:
  - line of *'s
  - filename
  - programmer's name
  - blank line
  - program description
  - line of *'s
  - blank line



Class Name / Source Code Filename

- All Java programs must be enclosed in a *class*. Think of a class as the name of the program.
- The name of the Java program's file must match the name of the Java program's class (except that the filename has a `.java` extension added to it).
- Proper style dictates that class names start with an uppercase first letter.
- Since Java is *case-sensitive*, that means the filename should also start with an uppercase first letter.
- Case-sensitive means that the Java compiler does distinguish between lowercase and uppercase letters.



`main` Method Heading

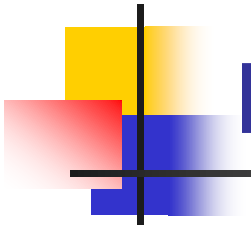
- Memorize (and always use) `public class` prior to your class name. For example:

```
public class Hello
```

- Inside your class, you must include one or more *methods*.
- A method is a group of instructions that solves one task. Later on, we'll have larger programs and they'll require multiple methods because they'll solve multiple tasks. But for now, we'll work with small programs that need only one method - the `main` method.
- Memorize (and always use) this `main` method heading:

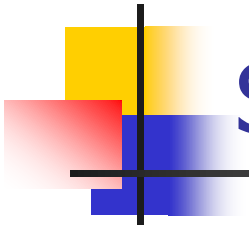
```
public static void main(String[] args)
```

- When a program starts, the computer looks for the `main` method and begins execution with the first statement after the `main` method heading.



Braces

- Use braces, { }, to group things together.
- For example, in the Hello World program, the top and bottom braces group the contents of the entire class, and the interior braces group the contents of the `main` method.
- Proper style dictates:
  - Place an opening brace on a line by itself in the same column as the first character of the previous line.
  - Place a closing brace on a line by itself in the same column as the opening brace.

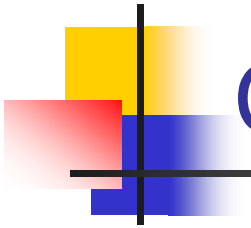


System.out.println

- To generate output, use `System.out.println()`.
  - For example, to print the hello message, we did this:

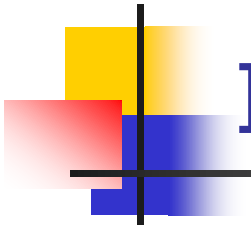
    ```
    System.out.println("Hello, world!");
    ```

  - Note:
    - Put the printed item inside the parentheses.
    - Surround strings with quotes.
    - Put a semicolon at the end of a `System.out.println` statement.
- What's the significance of the `ln` in `println`?



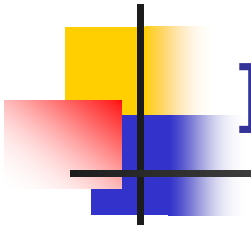
Compilation and Execution

- To create a Java program that can be run on a computer, submit your Java source code to a *compiler*. We say that the compiler *compiles* the source code. In compiling the source code, the compiler generates a bytecode program that can be run by the computer's JVM (Java Virtual Machine).
- Java source code filename = <*class-name*> + `.java`
- Java bytecode filename = <*class-name*> + `.class`



Identifiers

- Identifier = the technical term for a name in a programming language
- Identifier examples –
  - class name identifier: `Hello`
  - method name identifier: `main`
  - variable name identifier: `height`
- Identifier naming rules:
  - Must consist entirely of letters, digits, dollar signs ($), and/or underscore (_) characters.
  - The first character must not be a digit.
  - If these rules are broken, your program won't compile.



Identifiers

- Identifier naming conventions (style rules):
  - If these rules are broken, it won't affect your program's ability to compile, but your program will be harder to understand and you'll lose style points on your homework.
  - Use letters and digits only, not $'s or _'s.
  - All letters must be lowercase except the first letter in the second, third, etc. words. For example:

    ```
    firstName, x, daysInMonth
    ```

  - Addendum to the above rule – for class names, the first letter in every word (even the first word) must be uppercase. For example:

    ```
    StudentRecord, WorkShiftSchedule
    ```

  - Names must be descriptive.

Variables

- A variable can hold only one type of data. For example, an integer variable can hold only integers, a string variable can hold only strings, etc.
- How does the computer know which type of data a particular variable can hold?
  - Before a variable is used, its *type* must be *declared* in a *declaration* statement.
- Declaration statement syntax:

```
<type> <list of variables separated by commas>;
```

- Example declarations:

```
String firstName;    // student's first name
String lastName;     // student's last name
int studentId;
int row, col;
```

Style: comments must be aligned.

Assignment Statements

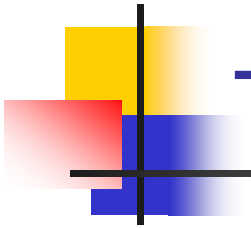
- Java uses the single equal sign (=) for assignment statements.
- In the below code fragment, the first assignment statement assigns the value 50000 into the variable `salary`.

```
int salary;
String bonusMessage;
salary = 50000;
bonusMessage = "Bonus = $" + (.02 * salary);
```

Commas are not allowed in numbers.

string concatenation

- Note the + operator in the second assignment statement. If a + operator appears between a string and something else (e.g., a number or another string), then the + operator performs string *concatenation*. That means that the JVM appends the item at the right of the + to the item at the left of the +, forming a new string.



Tracing

- Trace this code fragment:

```
int salary;
String bonusMessage;
salary = 50000;
bonusMessage = "Bonus = $" + (.02 * salary);
System.out.println(bonusMessage);
```

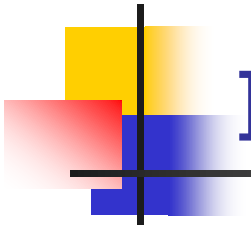
salary	bonusMessage	output

- When you trace a declaration statement, write a ? in the declared variable's column, indicating that the variable exists, but it doesn't have a value yet.

Program Template

- In this chapter's slides, all of the code fragment examples can be converted to complete programs by plugging them into the <*method-body*> in this program template:

```
/*************************************************
* Test.java
* <author>
*
* <description>
*************************************************/

public class Test
{
  public static void main(String[] args)
  {
    <method-body>
  }
} // end class Test
```



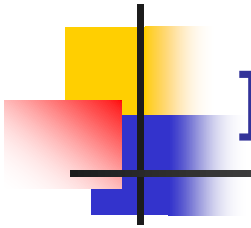
Initialization Statements

- Initialization statement:
  - When you assign a value to a variable as part of the variable's declaration.
- Initialization statement syntax:

```
<type> <variable> = <value>;
```

- Example initializations:

```
int totalScore = 0;         // sum of all bowling scores
int maxScore = 300;         // default maximum bowling score
```



Initialization Statements

- Example initializations (repeated from previous slide):

```
int totalScore = 0;         // sum of all bowling scores
int maxScore = 300;         // default maximum bowling score
```

- Here's an alternative way to do the same thing using declaration and assignment statements (instead of using initialization statements):

```
int totalScore;     // sum of all bowling scores
int maxScore;       // default maximum bowling score
totalScore = 0;
maxScore = 300;
```

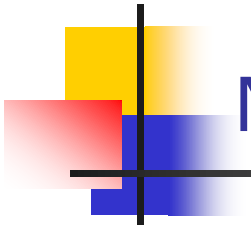
- It's OK to use either technique and you'll see it done both ways in the real world.

Numeric Data Types – `int`, `long`

- Variables that hold whole numbers (e.g., 1000, -22) should normally be declared with one of these integer data types – `int`, `long`.
- Range of values that can be stored in an `int` variable:
  - ≈ -2 billion to +2 billion
- Range of values that can be stored in a `long` variable:
  - ≈ -9×10^{18} to $+9 \times 10^{18}$
- Example integer variable declarations:

```
int studentId;
long satelliteDistanceTraveled;
```

- Recommendation: Use smaller types for variables that will never need to hold large values.

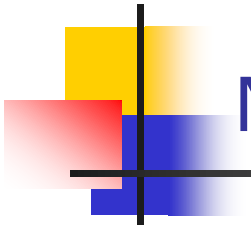


Numeric Data Types – `float`, `double`

- Variables that hold decimal numbers (e.g., -1234.5, 3.1452) should be declared with one of these floating-point data types – `float`, `double`.
- Example code:

```
float gpa;
double bankAccountBalance;
```

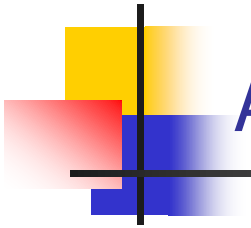
- The `double` type stores numbers using 64 bits whereas the `float` type stores numbers using only 32 bits. That means that double variables are better than float variables in terms of being able to store bigger numbers and numbers with more significant digits.



Numeric Data Types – `float`, `double`

- Recommendation:
  - You should normally declare your floating point variables with the `double` type rather than the `float` type.
  - In particular, don't use `float` variables when there are calculations involving money or scientific measurements. Those types of calculations require considerable accuracy and float variables are not very accurate.

- Range of values that can be stored in a `float` variable:
  - $\approx$ $-3.4*10^{38}$ to $+3.4*10^{38}$
- Range of values that can be stored in a `double` variable:
  - $\approx$ $-3.4*10^{308}$ to $+3.4*10^{308}$
- You can rely on 15 significant digits for a `double` variable, but only 6 significant digits for a `float` variable.



Assignments Between Different Types

- Assigning an integer value into a floating-point variable works just fine. Note this example:

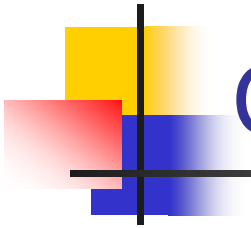
```
double bankAccountBalance = 1000;
```

- On the other hand, assigning a floating-point value into an integer variable is like putting a large object into a small box. By default, that's illegal. For example, this generates a compilation error:

```
int temperature = 26.7;
```

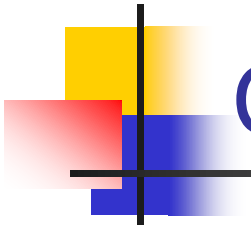
- This statement also generates a compilation error:

```
int count = 0.0;
```



Constants

- A constant is a fixed value. Examples:
  - 8, -45, 2000000 : integer constants
  - -34.6, .009, 8. : floating point constants
  - "black bear", "hi" : string constants
- The default type for an integer constant is `int` (not `long`).
- The default type for a floating point constant is `double` (not `float`).



Constants

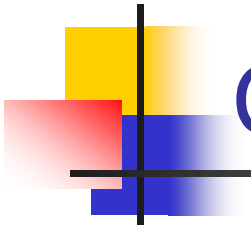
- This example code generates compilation errors. Where and why?

```
float gpa = 2.30;
float mpg;
mpg = 50.5;
```

- Possible Solutions:
  - Always use `double` variables instead of `float` variables.

    or
  - To explicitly force a floating point constant to be `float`, use an `f` or `F` suffix. For example:

```
float gpa = 2.30f;
float mpg;
mpg = 50.5F;
```



Constants

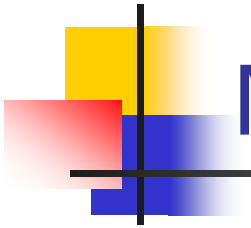
- Constants can be split into two categories: hard-coded constants and named constants.
- The constants we've covered so far can be referred to as *hard-coded constants*. A hard-coded constant is an explicitly specified value. For example, in this assignment statement, 299792458.0 is a hard-coded constant:

division operator

```
propagationDelay = cableLength / 299792458.0;
```

- A *named constant* is a constant that has a name associated with it. For example, in this code fragment, SPEED_OF_LIGHT is a named constant:

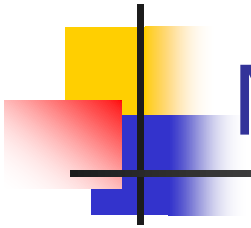
```
final double SPEED_OF_LIGHT = 299792458.0; // in m/s
...
propagationDelay = cableLength / SPEED_OF_LIGHT;
```



Named Constants

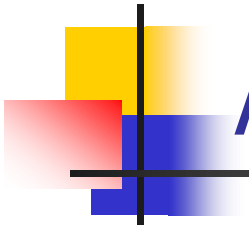
- The reserved word `final` is a modifier – it modifies SPEED_OF_LIGHT so that its value is fixed or "final."
- All named constants use the `final` modifier.
- The `final` modifier tells the compiler to generate an error if your program ever tries to change the `final` variable's value at a later time.

- Standard coding conventions suggest that you capitalize all characters in a named constant and use an underscore to separate the words in a multiple-word named constant.



Named Constants

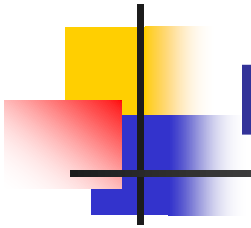
- There are two main benefits of using named constants:
  1. Using named constants leads to code that is more understandable.
  2. If a programmer ever needs to change a named constant's value, the change is easy – find the named constant initialization at the top of the method and change the initialization value. That implements the change automatically everywhere within the method.



Arithmetic Operators

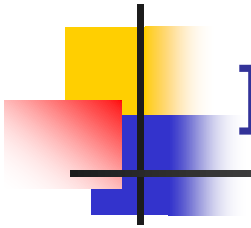
- Java's +, -, and * arithmetic operators perform addition, subtraction, and multiplication in the normal fashion.
- Java performs division differently depending on whether the numbers/operands being divided are integers or floating-point numbers.
- When the Java Virtual Machine (JVM) performs division on floating-point numbers, it performs "calculator division." We call it "calculator division" because Java's floating-point division works the same as division performed by a standard calculator. For example, if you divide 7.0 by 2.0 on your calculator, you get 3.5. Likewise, this code fragment prints 3.5:

```
System.out.println(7.0 / 2.0);
```



Floating-Point Division

- This next line says that 7.0 / 2.0 "evaluates to" 3.5:
  7.0 / 2.0 ⇒ 3.5
- This next line asks you to determine what 5 / 4. evaluates to:
  5 / 4. ⇒ ?
- 5 is an `int` and 4. is a `double`. This is an example of a *mixed expression*. A mixed expression is an expression that contains operands with different data types.
- `double` values are considered to be more complex than `int` values because `double` values contain a fractional component.
- Whenever there's a mixed expression, the JVM temporarily promotes the less-complex operand's type so that it matches the more-complex operand's type, and then the JVM applies the operator.
- In the 5 / 4. expression, the 5 gets promoted to a `double` and then floating-point division is performed. The expression evaluates to 1.25.



Integer Division

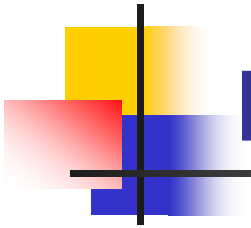
- There are two ways to perform division on integers:
  - The / operator performs "grade school" division and generates the quotient. For example:

    7 / 2 $\Rightarrow$?

  - The % operator (called the *modulus operator*) also performs "grade school" division and generates the remainder. For example:

    7 % 2 $\Rightarrow$?

    8 % 12 $\Rightarrow$?



Expression Evaluation Practice

- Given these initializations:

```
int a = 5, b = 2;
double c = 3.0;
```

- Use Chapter 3's operator precedence table to evaluate the following expressions:

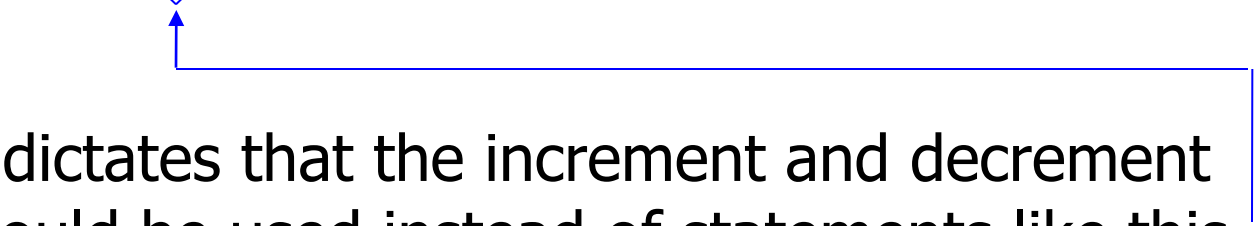
```
(c + a / b) / 10 * 5
```

```
(0 % a) + c + (0 / a)
```

Increment and Decrement Operators

- Use the increment operator (++) operator to increment a variable by 1. Use the decrement operator (--) to decrement a variable by 1.
- Here's how they work:

```
x++;     ≡     x = x + 1;
x--;     ≡     x = x - 1;
```



- Proper style dictates that the increment and decrement operators should be used instead of statements like this.

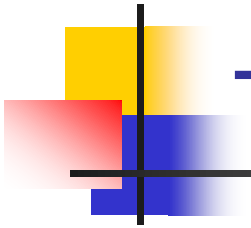
Compound Assignment Operators

- The compound assignment operators are:
  - +=, -=, *=, /=, %=
- The variable is assigned an updated version of the variable's original value.
- Here's how they work:

```
x += 3;        ≡    x = x + 3;
x -= 4;        ≡    x = x - 4;
```

Repeat the variable on both sides of the "="

- Proper style dictates that compound assignment operators should be used instead of statements like this



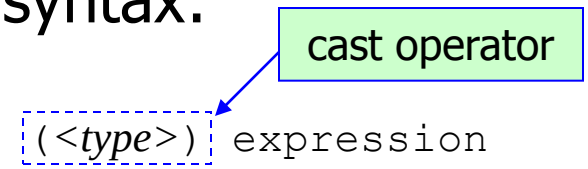
Tracing Practice

- Trace this code fragment:

```
int a = 4, b = 6;
double c = 2.0;
a -= b;
b--;
c++;
c *= b;
System.out.println("a + b + c = " + (a + b + c));
```

Type Casting

- In writing a program, you'll sometimes need to convert a value to a different data type. The cast operator performs such conversions. Here's the syntax:



- Suppose you've got a variable named `interest` that stores a bank account's interest as a `double`. You'd like to extract the dollars portion of the interest and store it in an `int` variable named `interestInDollars`. To do that, use the `int` cast operator like this:

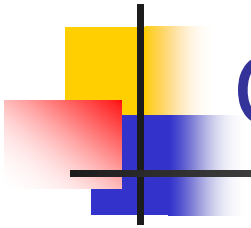
```
interestInDollars = (int) interest;
```

Type Casting

- If you ever need to cast more than just a single value or variable (i.e., you need to cast an expression), then make sure to put parentheses around the entire thing that you want casted. Note this example:

```
double interestRate;
double balance;
int interestInDollars;
...
interestInDollars = (int) (balance * interestRate);
```

Parentheses are necessary here.



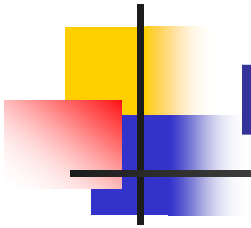
Character Type - `char`

- A `char` variable holds a single character.
- A `char` constant is surrounded by single quotes.
- Example `char` constants:
  - `'B','1',':'`
- Example code fragment:

```
char first, middle, last;
first = 'J';
middle = 'S';
last = 'D';
System.out.println("Hello, " + first + middle +
  last + '!');
```

- What does this code fragment print?

Escape Sequences

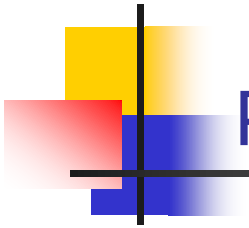


- *Escape sequences* are `char` constants for hard-to-print characters such as the enter character and the tab character.
- An escape sequence is comprised of a backslash (\) and another character.
- Common escape sequences:
  - \n newline – go to first column in next line
  - \t move the cursor to the next tab stop
  - \\ print a backslash
  - \" print a double quote
  - \' print a single quote

- Provide a one-line print statement that prints these tabbed column headings followed by two blank lines:

```
ID    NAME
```

- Note that you can embed escape sequences inside strings the same way that you would embed any characters inside a string. For example, provide an improved one-line print statement for the above heading.
- Why is it called an "escape" sequence?



Primitive Variables vs. Reference Variables

- There are two basic categories of variables in Java – primitive variables and reference variables.
- Primitive variables hold only one piece of data. Primitive variables are declared with a primitive type and those types include:
  - `int`, `long` (integer types)
  - `float`, `double` (floating point types)
  - `char` (character type)
- Reference variables are more complex - they can hold a group of related data. Reference variables are declared with a reference type and here are some example reference types:
  - `String`, `Calendar`, programmer-defined classes

Reference types start with an uppercase first letter.

String Basics

- Example code for basic string manipulations:

```
String s1;
String s2 = "and I say hello";
s1 = "goodbye";
s1 = "You say " + s1;
s1 += ", " + s2 + '.';
System.out.println(s1);
```

declaration

initialization

assignment

concatenation, then assignment

concatenation, then compound assignment

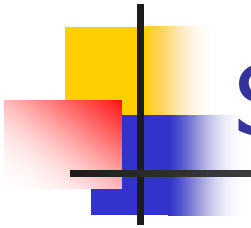
- Trace the above code.

String Methods

- `String`'s `charAt` method:
  - Returns the character in the given string at the specified position.
  - The positions of the characters within a string are numbered starting with position zero.
  - What's the output from this example code?

```
String animal = "cow";
System.out.println("Last character: " + animal.charAt(2));
```

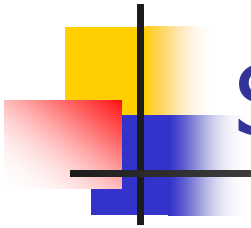
To use a method, include the reference variable, dot, method name, parentheses, and argument(s).



String Methods

- `String`'s `length` method:
  - Returns the number of characters in the string.
  - What's the output from this code fragment?

```
String s = "hi";
System.out.println(s.length());
```



String Methods

- To compare strings for equality, use the `equals` method. Use `equalsIgnoreCase` for case-insensitive equality.
- Trace this program:

```
public class Test
{
  public static void main(String[] args)
  {
    String animal1 = "Horse";
    String animal2 = "Fly";
    String newCreature;
    newCreature = animal1 + animal2;

    System.out.println(newCreature.equals("HorseFly"));
    System.out.println(newCreature.equals("horsefly"));
    System.out.println(newCreature.equalsIgnoreCase("horsefly"));
  } // end main
} // end class Test
```

Input – the Scanner Class

- Sun provides a pre-written class named `Scanner`, which allows you to get input from a user.
- To tell the compiler you want to use the `Scanner` class, insert the following `import` statement at the very beginning of your program (right after your prologue section and above the `main` method):

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

- At the beginning of your `main` method, insert this initialization statement:

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

- After declaring `stdIn` as shown above, you can read and store a line of input by calling the `nextLine` method like this:

```
<variable> = stdIn.nextLine();
```

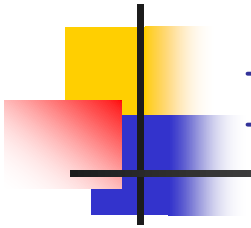
Input – the Scanner Class

```
/*********************************************************
* FriendlyHello.java
* Dean & Dean
*
* This program displays a personalized Hello greeting.
*********************************************************/

import java.util.Scanner;

public class FriendlyHello
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String name;
    System.out.print("Enter your name: ");
    name = stdIn.nextLine();
    System.out.println("Hello " + name + "!");
  } // end main
} // end class FriendlyHello
```

These two statements create a keyboard-input connection.

This gets a line of input.

Use the `print` method (no "ln") for most prompts.



`Input` – the `Scanner` Class

- In addition to the `nextLine` method, the `Scanner` class contains quite a few other methods that get different forms of input. Here are some of those methods:

`nextInt()`

Skip leading whitespace until an `int` value is found. Return the `int` value.

`nextLong()`

Skip leading whitespace until a `long` value is found. Return the `long` value.

`nextFloat()`

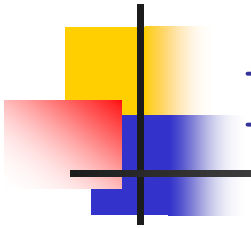
Skip leading whitespace until a `float` value is found. Return the `float` value.

`nextDouble()`

Skip leading whitespace until a `double` value is found. Return the `double` value.

`next()`

Skip leading whitespace until a token is found. Return the token as a `String` value.



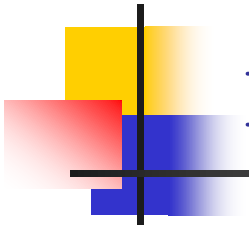
`Input` – the `Scanner` Class

- What is *whitespace*?
  - Whitespace refers to all characters that appear as blanks on a display screen or printer. This includes the space character, the tab character, and the newline character.
  - The newline character is generated with the enter key.
  - *Leading* whitespace refers to whitespace characters that are at the left side of the input.
- What is a *token*?
  - A token is a sequence of non-whitespace characters.
- What happens if the user provides invalid input for one of `Scanner`'s method calls?
  - The JVM prints an error message and stops the program.
  - For example, 45g and 45.0 are invalid inputs if `nextInt()` is called.

Input – the Scanner Class

- Here's a program that uses Scanner's `nextDouble` and `nextInt` methods:

```
import java.util.Scanner;

public class PrintPO
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double price;  // price of purchase item
    int qty;       // number of items purchased

    System.out.print("Price of purchase item: ");
    price = stdIn.nextDouble();
    System.out.print("Quantity: ");
    qty = stdIn.nextInt();
    System.out.println("Total purchase order = $" + price * qty);
  } // end main
} // end class PrintPO
```



Input – the Scanner Class

- Here's a program that uses Scanner's next method:

```
import java.util.Scanner;

public class PrintInitials
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String first;  // first name
    String last;   // last name

    System.out.print(
      "Enter first and last name separated by a space: ");
    first = stdIn.next();
    last = stdIn.next();
    System.out.println("Your initials are " +
      first.charAt(0) + last.charAt(0) + ".");
  } // end main
} // end class PrintInitials
```