Java Basics

Rahul Deodhar
rahuldeodhar@gmail.com
www.rahuldeodhar.com
+91 9820213813

Chapter 3 - 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, <u>but</u> 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:
  - ≈ -9x10^{18} to +9x10^{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:

Repeat the variable on both sides of the "="

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

- 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:

cast operator

```
(<type>) expression
```

- 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
```

“Static” Operator

- Static Operator used in
  - Variables
    - Makes the variable generic/common for the entire class
  - Methods
    - Method belongs to class rather than object
    - Can be invoked without creating instance of the class
    - Using <class_name>.<method_name>
    - Cannot use non-static data members
    - Cannot use other non-static methods
    - Cannot use “this” and “super”
  - Block
    - Used to initialize static members
    - Executes before main method at the time of class loading.

`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
```

Chapter 4 - Control Statements

- Conditions
- `if` Statement
- `&&` Logical Operator
- `||` Logical Operator
- `!` Logical Operator
- `switch` Statement
- `while` Loop
- `do` Loop
- `for` Loop
- Loop Comparison
- Nested Loops
- Boolean Variables
- Input Validation
- Boolean Logic
- Expression Evaluation Practice

Conditions

- Throughout this chapter, you'll see if statements and loop statements where *conditions* appear within a pair of parentheses, like this:

```
if (<condition>)
{
  …
}

while (<condition>)
{
  …
}
```

- Typically, each condition involves some type of comparison and the comparisons use comparison operators….

Conditions

- Here are Java's comparison operators:
  ==, !=, <, >, <=, >=
- Each comparison operator evaluates to either true or false.
- ==
  - Tests two operands for equality.
  - 3 == 3 evaluates to true
  - 3 == 4 evaluates to false
  - Note that == uses two equal signs, not one!
- !=
  - Tests two operands for inequality.
  - The != operator is pronounced “not equal.”
- The <, >, <=, and >= operators work as expected.

`if` Statement

- Use an `if` statement if you need to ask a question in order to determine what to do next.
- There are three forms for an `if` statement:
  - `if` by itself
    - Use for problems where you want to do something or nothing.
  - `if, else`
    - Use for problems where you want to do one thing or another thing.
  - `if, else if`
    - Use for problems where you want to do one thing out of three or more choices.

`if` Statement

<u>pseudocode syntax</u>

- `if` by itself:

```
if <condition>
   <statement(s)>
```

- `if, else`:

```
if <condition>
   <statement(s)>
else
   <statement(s)>
```

<u>Java syntax</u>

- `if` by itself:

```
if (<condition>)
{
   <statement(s)>
}
```

- `if, else`:

```
if (<condition>)
{
   <statement(s)>
}
else
{
   <statement(s)>
}
```

`if` Statement

pseudocode syntax

`if, else if:`

```
if <condition>
    <statement(s)>
else if <condition>
    <statement(s)>
 .  }
 .  }  <- more else if's here (optional)
 .  }
else                 }
    <statement(s)>   }  <- optional
```

Java syntax

`if, else if, else:`

```
if (<condition>)
{
    <statement(s)>
}
else if (<condition>)
{
    <statement(s)>
}
 .  }
 .  }  <- more else if's here (optional)
 .  }
else                 }
{                    }
    <statement(s)>   }  <- optional
}                    }
```

`if` Statement

- Write a complete program that prompts the user to enter a sentence and then prints an error message if the last character is not a period.

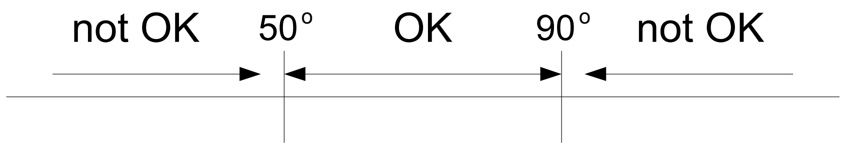
sample session:

Italics indicates input. Never *hardcode* (include) input as part of your source code!!!

```
Enter a sentence:
Permanent good can never be the outcome of violence
Invalid entry – your sentence needs a period!
```

& & Logical Operator

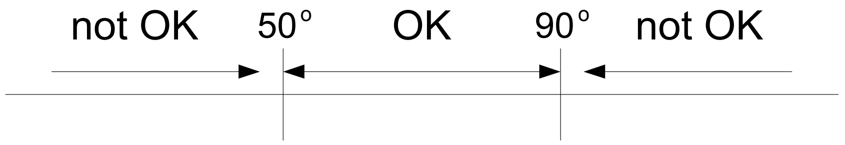
- Suppose you want to print "OK" if the temperature is between 50 and 90 degrees and print "not OK" otherwise.



- Here's the pseudocode solution:

```
if temp ≥ 50 and ≤ 90
    print "OK"
else
    print "not OK"
```

&& Logical Operator



- And here's the solution using Java:

```
if (temp >= 50 && temp <= 90)
{
    System.out.println("OK");
}
else
{
    System.out.println("not OK");
}
```

- In Java, if two criteria are required for a condition to be satisfied (e.g., temp >= 50 and temp <= 90), then separate the two criteria with the `&&` (and) operator. If both criteria use the same variable (e.g., `temp`), you must include the variable on both sides of the `&&`.

&& Logical Operator

- The program on the next slide determines whether fans at a basketball game win free french fries. If the home team wins and scores at least 100 points, then the program prints this message:

```
Fans: Redeem your ticket stub for a free order of french
fries at Yummy Burgers.
```

- On the next slide, replace *<insert code here>* with appropriate code.

& & Logical Operator

```
/************************************************************
* FreeFries.java
* Dean & Dean
*
* This program reads points scored by the home team
* and the opposing team and determines whether the
* fans win free french fries.
************************************************************/

import java.util.Scanner;

public class FreeFries
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int homePts;        // points scored by home team
    int opponentPts;    // points scored by opponents
    System.out.print("Home team points scored: ");
    homePts = stdIn.nextInt();
    System.out.print("Opposing team points scored: ");
    opponentPts = stdIn.nextInt();

    <insert code here>

  } // end main
} // end class FreeFries
```

`||` Logical Operator

- Provide code that prints "bye" if a `response` variable contains a lowercase or uppercase q (for quit). Here's a pseudocode implementation:

```
if response equals "q" or "Q"
    print "Bye"
```

- To implement "or" logic in Java, use `||` (the or operator). Here's the Java implementation:

```
if (response.equals("q") || response.equals("Q"))
{
  System.out.println("bye");
}
```

When using the || operator, if both criteria in the or condition use the same variable (e.g., `response`), you must include the variable on both sides of the ||.

| | Logical Operator

- It's a common bug to forget to repeat a variable that's part of an || (or &&) condition. This code generates a compilation error:

```
if (response.equals("q" || "Q"))
{
  System.out.println("bye");
}
```

- Another common bug is to use the == operator to compare strings for equality. This code compiles successfully, but it doesn't work properly:

```
if (response == "q" || response == "Q")
{
  System.out.println("bye");
}
```

| | Logical Operator

- As an alternative to using the || operator with two `equals` method calls, you could use an `equalsIgnoreCase` method call like this:

```
if (response.equalsIgnoreCase("q"))
{
  System.out.println("Bye");
}
```

`!` Logical Operator

- The `!` (not) operator reverses the truth or falsity of a condition.
- For example, suppose that a `char` variable named `reply` holds a *q* (lowercase or uppercase) if the user wants to quit, or some other character if the user wants to continue. To check for "some other character" (i.e., not a *q* or *Q*), use the ! operator like this:

```
if (!(reply == 'q' || reply == 'Q'))
{
  System.out.println("Let's get started....");
  ...
```

`switch` Statement

- When to use a `switch` statement:
  - If you need to do one thing from a list of multiple possibilities.
- Note that the `switch` statement can always be replaced by an `if`, `else if`, `else` statement, but the `switch` statement is considered to be more elegant. (Elegant code is easy to understand, easy to update, robust, reasonably compact, and efficient.)
- Syntax:

```
switch (<controlling-expression>)
{
  case <constant1>:
    <statements>;
    break;
  case <constant2>:
    <statements>;
    break;
  ...
  default:
    <statements>;
} // end switch
```

`switch` Statement

- How the `switch` statement works:
  - Jump to the `case` constant that matches the controlling expression's value (or jump to the `default` label if there are no matches) and execute all subsequent statements until reaching a `break`.
  - The `break` statement causes a jump out of the switch statement (below the "}").
  - Usually, `break` statements are placed at the end of every `case` block. However, that's not a requirement and they're sometimes omitted for good reasons.
  - Put a **:** after each `case` constant.
  - Even though statements following the `case` constants are indented, { }'s are not necessary.
  - The controlling expression should evaluate to either an `int` or a `char`.
  - Proper style dictates including "// end switch" after the `switch` statement's closing brace.

switch Statement

- Given this code fragment:

```
i = stdIn.nextInt();
switch (i)
{
  case 1:
    System.out.print("A");
    break;
  case 2:
    System.out.print("B");
  case 3: case 4:
    System.out.print("C-D");
    break;
  default:
    System.out.print("E-Z");
} // end switch
```

- If input = 1, what's the output?
- If input = 2, what's the output?
- If input = 3, what's the output?
- If input = 4, what's the output?
- If input = 5, what's the output?

`switch` Statement

- Write a program that reads in a ZIP Code and uses the first digit to print the associated geographic area:

if zip code begins with	print this message
0, 2, 3	*<zip>* `is on the East Coast.`
4-6	*<zip>* `is in the Central Plains area.`
7	*<zip>* `is in the South.`
8-9	*<zip>* `is in the West.`
other	*<zip>* `is an invalid ZIP Code.`

- Note: *<zip>* represents the entered ZIP Code value.

`while` Loop

- Use a loop statement if you need to do the same thing repeatedly.

<u>pseudocode syntax</u>

```
while <condition>
   <statement(s)>
```

<u>Java syntax</u>

```
while (<condition>)
{
   <statement(s)>
}
```

`while` Loop

- Write a `main` method that finds the sum of user-entered integers where -99999 is a sentinel value.

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  int sum = 0;         // sum of user-entered values
  int x;               // a user-entered value
  System.out.print("Enter an integer (-99999 to quit): ");
  x = stdIn.nextInt();
  while (x != -99999)
  {
    sum = sum + x;
    System.out.print("Enter an integer (-99999 to quit): ");
    x = stdIn.nextInt();
  }
  System.out.println("The sum is " + sum);
} // end main
```

`do` Loop

- When to use a `do` loop:
  - If you know that the repeated thing will always have to be done at least one time.
- Syntax:

```
do
{
   <statement(s)>
} while (<condition>);
```

- Note:
  - The condition is at the bottom of the loop (in contrast to the `while` loop, where the condition is at the top of the loop).
  - The compiler requires putting a ";" at the very end, after the `do` loop's condition.
  - Proper style dictates putting the "while" part on the same line as the "}"

`do` Loop

- Problem description:
  - As part of an architectural design program, write a `main` method that prompts the user to enter length and width dimensions for each room in a proposed house so that total floor space can be calculated for the entire house.
  - After each length/width entry, ask the user if there are any more rooms.
  - Print the total floor space.

`for` Loop

- When to use a `for` loop:
  - If you know the exact number of loop iterations before the loop begins.
- For example, use a `for` loop to:
  - Print this countdown from 10.

    Sample session:

    ```
    10 9 8 7 6 5 4 3 2 1 Liftoff!
    ```
  - Find the factorial of a user-entered number.

    Sample session:

    ```
    Enter a whole number: 4
    4! = 24
    ```

`for` Loop

`for` loop syntax

```
for (<initialization>; <condition>; <update>)
{
    <statement(s)>
}
```

`for` loop example

```
for (int i=10; i>0; i--)
{
  System.out.print(i + " ");
}
System.out.println("Liftoff!");
```

- `for` loop semantics:
  - Before the start of the first loop iteration, execute the initialization component.
  - At the top of each loop iteration, evaluate the condition component:
    - If the condition is true, execute the body of the loop.
    - If the condition is false, terminate the loop (jump to the statement below the loop's closing brace).
  - At the bottom of each loop iteration, execute the update component and then jump to the top of the loop.

`for` Loop

- Trace this code fragment with an input value of 3.

```
Scanner stdIn = new Scanner(System.in);
int number;                 // user entered number
double factorial = 1.0; // factorial of user entry

System.out.print("Enter a whole number: ");
number = stdIn.nextInt();

for (int i=2; i<=number; i++)
{
  factorial *= i;
}

System.out.println(number + "! = " + factorial);
```

`for` loop *index variables* are often, but not always, named i for "index."

Declare `for` loop *index variables* within the `for` loop heading.

`for` Loop

- Write a `main` method that prints the squares for each odd number between 1 and 99.
- Output:

```
1
9
25
49
81
…
```

Loop Comparison

When to use / **Template**

`for` loop:

If you know, prior to the start of loop, how many times you want to repeat the loop.

```
for (int i=0; i<max; i++)
{
    <statement(s)>
}
```

`do` loop:

If you always need to do the repeated thing at least one time.

```
do
{
    <statement(s)>
    <prompt - do it again (y/n)?>
} while (<response == 'y'>);
```

`while` loop:

If you can't use a `for` loop or a `do` loop.

```
<prompt - do it (y/n)?>
while (<response == 'y'>)
{
    <statement(s)>
    <prompt - do it again (y/n)?>
}
```

Nested Loops

- Nested loops = a loop within a loop.
- Example – Write a program that prints a rectangle of characters where the user specifies the rectangle's height, the rectangle's width, and the character's value.

<u>Sample session</u>:

```
Enter height: 4
Enter width: 3
Enter character: <
<<<
<<<
<<<
<<<
```

Boolean Variables

- Programs often need to keep track of the state of some condition.
- For example, if you're writing a program that simulates the operations of a garage door opener, you'll need to keep track of the state of the garage door's direction - is the direction up or down? You need to keep track of the direction "state" because the direction determines what happens when the garage door opener's button is pressed. If the direction state is up, then pressing the garage door button causes the direction to switch to down. If the direction state is down, then pressing the garage door button causes the direction to switch to up.
- To implement the state of some condition, use a `boolean` *variable*.

Boolean Variables

- A `boolean` variable is a variable that:
  - Is declared to be of type `boolean`.
  - Holds the value `true` or the value `false`.
- Boolean variables are good at keeping track of the state of some condition when the state has one of two values. For example:

Values for the state of a garage door opener's direction	Associated values for a `boolean` variable named `upDirection`
up	true
down	false

Boolean Variables

- This code fragment initializes an `upDirection` variable to true and shows how to toggle its value within a loop.

```
boolean upDirection = true;
do
{
   ...
   upDirection = !upDirection;
   ...
} while (<user presses the garage door opener button>);
```

If `upDirection` holds the value `true`, this statement changes it to `false`, and vice versa.

Boolean Variables

```
import java.util.Scanner;

public class GarageDoor
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;                  // user's entry - enter key or q
    boolean upDirection = true; // Is the current direction up?
    boolean inMotion = false;   // Is garage door currently moving?

    System.out.println("GARAGE DOOR OPENER SIMULATOR\n");

    do
    {
      System.out.print("Press Enter, or enter 'q' to quit: ");
      entry = stdIn.nextLine();

      if (entry.equals(""))    // pressing Enter generates ""
      {
        inMotion = !inMotion;  // button toggles motion state
```

Boolean Variables

```
            if (inMotion)
            {
              if (upDirection)
              {
                System.out.println("moving up");
              }
              else
              {
                System.out.println("moving down");
              }
            }
            else
            {
              System.out.println("stopped");
              upDirection = !upDirection;  // direction reverses at stop
            }
          } // end if entry = ""
        } while (entry.equals(""));
      } // end main
    } // end GarageDoor class
```

Input Validation

- `boolean` variables are often used for *input validation*.
- Input validation is when a program checks a user's input to make sure it's valid, i.e., correct and reasonable. If it's valid, the program continues. If it's invalid, the program enters a loop that warns the user about the erroneous input and then prompts the user to re-enter.
- In the `GarageDoor` program, note how the program checks for an empty string (which indicates the user wants to continue), but it doesn't check for a q.

Input Validation

- To add input validation to the `GarageDoor` program, replace the `GarageDoor` program's prompt with the following code. It forces the user to press Enter or enter a q or Q.

```
validEntry = false;
do
{
  System.out.print("Press Enter, or enter 'q' to quit: ");
  entry = stdIn.nextLine();
  if (entry.equals("") || entry.equalsIgnoreCase("q"))
  {
    validEntry = true;
  }
  else
  {
    System.out.println("Invalid entry.");
  }
} while (validEntry == false);
```

What is a more elegant implementation for this?

Boolean Logic

- *Boolean logic* (= *Boolean algebra*) is the formal logic that determines how conditions are evaluated.
- The building blocks for Boolean logic are things that you've already seen - the logical operators &&, ||, and !.
- Logical operator review:
  - For the && operator, both sides need to be true for the whole thing to be true.
  - For the || operator, only one side needs to be true for the whole thing to be true.
  - The ! operator reverses the truth or falsity of something.

Expression Evaluation Practice

- Assume:

```
boolean ok = false;
double x = 6.5, y = 10.0;
```

- Evaluate these expressions:

```
(x != 6.5) || !ok



true && 12.0 < x + y
```

Chapter 5 - Using Pre-Built Methods

- The API Library
- API Headings
- `Math` Class
- Wrapper Classes for Primitive Types
- Lottery Example
- `String` Methods:
  - `substring`
  - `indexOf`
  - `lastIndexOf`
- Formatted Output with the `printf` Method

The API Library

- When working on a programming problem, you should normally check to see if there are pre-built classes that meet your program's needs.
- If there are such pre-built classes, then use those classes (don't "reinvent the wheel"). For example:
  - User input is a rather complicated task. The `Scanner` class handles user input. Whenever you need user input in a program, use the `Scanner` class's input methods (rather than writing and using your own input methods).
  - Math calculations are sometimes rather complicated. The Math class handles math calculations. Whenever you need to perform non-trivial math calculations in a program, use the `Math` class's methods (rather than writing and using your own math methods).

The API Library

- Java's pre-built classes are stored in its *class library*, which is more commonly known as the *Application Programming Interface* (API) library. See http://download.oracle.com/javase/6/docs/api/.
- Java's API classes are not part of the core Java language. For a program to use an API class, the class first needs to be loaded/ imported into the program. For example, to use the `Scanner` class, include this at the top of your program:

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

- The `java.util` thing that precedes `Scanner` is called a package.
- A package is a group of classes.
- The `java.util` package contains quite a few general-purpose utility classes. The only one you'll need for now is the `Scanner` class.

The API Library

- Some classes are considered to be so important that the Java compiler automatically imports them for you. The automatically imported classes are in the `java.lang` package.
- The `Math` class is one of those classes, so there's no need for you to import the `Math` class if you want to perform math operations.
- The Java compiler automatically inserts this statement at the top of every Java program:

```
 import java.lang.*;
```

- The asterisk is a wild card and it means that all classes in the `java.lang` package are imported, not just the `Math` class.

API Headings

- To use an API class, you don't need to know the internals of the class; you just need to know how to "interface" with it.
- To interface with a class, you need to know how to use the methods within the class. For example, to perform input, you need to know how to use the `Scanner` class's methods - `next`, `nextLine`, `nextInt`, `nextDouble`, etc.
- To use a method, you need to know what type of argument(s) to pass to it and what type of value it returns.
- The standard way to show that information is to show the method's source code heading.

API Headings

- For example, here's the source code heading for the `Scanner` class's `nextInt` method:

```
public int nextInt()
```

The arguments that you pass to the method would go inside the parentheses (but no arguments are passed to the `nextInt` method).

The *return type* (`int` in this example) indicates the type of the value that's being returned from the method.

All the methods in the API library are `public`, which means that they are accessible from everywhere; i.e., the "public" can access them.

- And here's an example of calling the `nextInt` method:

```
int days = stdIn.nextInt();
```

`Math` Class

- Source code headings for API methods are commonly referred to as *API headings*.
- Here are the API headings for some of the more popular methods in the `Math` class:
  - `public static int abs(int num)`
    - Returns the absolute value of `num`.
  - `public static double abs(double num)`
    - Returns the absolute value of `num`.
  - `public static int max(int x, int y)`
    - Returns the larger value of `x` and `y`.
  - `public static double max(double x, double y)`
    - Returns the larger value of `x` and `y`.

`Math` Class

- `Math` class API headings (continued):
  - `public static int min(int x, int y)`
    - Returns the smaller value of `x` and `y`.
  - `public static double min(double x, double y)`
    - Returns the smaller value of `x` and `y`.
  - `public static double pow(double num, double power)`
    - Returns `num` raised to the specified `power`.
  - `public static double random()`
    - Returns a uniformly distributed value between 0.0 and 1.0, but not including 1.0.
  - `public static long round(double num)`
    - Returns the whole number that is closest to `num`.
  - `public static double sqrt(double num)`
    - Returns the square root of `num`.

`Math` Class

- Note the `static` modifier at the left of all the Math methods. All the methods in the `Math` class are `static` *methods* (also called *class methods*), which means they are called by prefacing the method's name with the name of the class in which they are defined. For example:

```
int position1 = 15, position2 = 18;
int distanceApart = Math.abs(position1 - position2);
```

Call `Math` methods by prefacing them with `Math` dot.

- Write a Java statement that updates x's value so x gets the absolute value of its original value.

`Math` Class

- It is legal to pass an integer value to a method that accepts a floating-point argument. Note the following example. Horton's Law says that the length of a river is related to the area drained by the river in accordance with this formula:

  length ≈ 1.4 (area)$^{0.6}$

- Here's how to implement Horton's Law in Java code:

```
int area = 10000;     // square miles drained
double riverLength = 1.4 * Math.pow(area, 0.6);
```

OK to pass an `int` (`area`), into `pow`, which accepts `double` arguments.

- A common use of computers is to model real-world activities that rely on random events.
- That's because computers are good at generating random numbers and being able to repeat the random events many, many times.

Math Class

- The Math class contains a named constant, PI.
  - Pi, written as π in math books, is the ratio of a circle's perimeter to its diameter.
  - It contains this double value: 3.14159265358979323846
  - It's a constant, which means its value is fixed. If you attempt to assign a value to it, you'll get a compilation error.
  - Just like Math's methods are class methods and they are accessed using the Math class name, Math's PI is a *class variable* and it is accessed using the Math class name. In other words, if you need pi, specify Math.PI.
- Complete this code fragment:

```
double radius = 3.0;
double volumeOfSphere =
```

Wrapper Classes For Primitive Types

- A *wrapper class* is a class that surrounds a relatively simple item in order to add functionality to the simple item.
- Here are wrapper classes for some of the Java primitive types:

Wrapper Class	Primitive Type
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Character`	`char`

- Note that the wrapper class names are the same as the primitive names except for the uppercase first letter. What are the exceptions to that rule?
- The wrapper classes are defined in the `java.lang` package. The Java compiler automatically imports all the classes in the `java.lang` package, so there's no need to import the wrapper classes explicitly.

Wrapper Classes For Primitive Types

- Most real-world Java programs use GUI I/O instead of text-based I/O. (GUI = graphical user interface. I/O = input/output.)
  - What is text-based I/O?
  - What is GUI I/O?
- With GUI programs, all numeric output is string based. So to display a number, you need to convert the number to a string prior to calling the GUI display method. All numeric input is string based, too. So to read a number in a GUI program, you first read the input as a string and then convert the string to a number.
- Here are string conversion methods provided by the numeric wrapper classes:

Wrapper Class	string → number	number → string
`Integer`	`Integer.parseInt(`*<string>*`)`	`Integer.toString(<#>)`
`Long`	`Long.parseLong(`*<string>*`)`	`Long.toString(<#>)`
`Float`	`Float.parseFloat(`*<string>*`)`	`Float.toString(<#>)`
`Double`	`Double.parseDouble(`*<string>*`)`	`Double.toString(<#>)`

Wrapper Classes For Primitive Types

- Conversion examples - strings to numbers:

```
String yearStr = "2011";
String scoreStr = "78.5";
int year = Integer.parseInt(yearStr);
double score = Double.parseDouble(scoreStr);
```

- Remember - to convert a string to a numeric type, use X.parseX where X is the numeric type you're interested in.
- Conversion examples - numbers to strings :

```
int year = 2011;
double score = 78.5;
String yearStr = Integer.toString(year);
String scoreStr = Double.toString(score);
```

Wrapper Classes For Primitive Types

- To find the largest and smallest possible values for a particular type, use the type's wrapper class and access the wrapper class's `MAX_VALUE` and `MIN_VALUE` named constants. For example:

```
Integer.MAX_VALUE
Double.MAX_VALUE
Long.MIN_VALUE
```

- Write a lottery program that prompts the user to guess a randomly generated number between 0 and the maximum `int` value. The user pays $1.00 for each guess and wins $1,000,000 if the guess is correct. The user enters a "q" to quit.

Lottery Example

```
import java.util.Scanner;

public class Lottery
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String input;
    int winningNumber =


    System.out.println("Want to win a million dollars?");
    System.out.println("If so, guess the winning number (a" +
      " number between 0 and " + Integer.MAX_VALUE + ").");
    do
    {
      System.out.print(
        "Insert $1.00 and enter your number or 'q' to quit: ");
      input = stdIn.nextLine();
```

Initialize `winningNumber` to a randomly chosen integer between 0 and the largest possible `int`.

Hint: Use `Math.ceil`.

Lottery Example

```
      if (input.equals("give me a hint"))     //  a back door
      {
        System.out.println("try: " + winningNumber);
      }
      else if (!input.equals("q"))
      {
        if (
        {
          System.out.println("YOU WIN!");
          input = "q"; // force winners to quit
        }
        else
        {
          System.out.println(
            "Sorry, good guess, but not quite right.");
        }
      } // end else if
    } while (!input.equals("q"));
    System.out.println("Thanks for playing. Come again!");
  } // end main
} // end Lottery class
```

Compare input with the winning number.

The `String` Class

- In Chapter 3, you saw several `String` methods - `charAt`, `length`, `equals`, and `equalsIgnoreCase`.
- Those methods are defined in the `String` class, along with quite a few other methods that help with string manipulation tasks.
- The `String` class is defined in the `java.lang` package. The Java compiler automatically imports all the classes in the `java.lang` package, so there's no need to import the `String` class explicitly.
- We'll now present several additional popular methods in the `String` class.

The `String` Class's `substring` Method

- Here are API headers and brief descriptions for two alternative forms of the `substring` method:
  - `public String substring(int beginIndex)`
    - Returns a string that is a subset of the calling-object string, starting at the `beginIndex` position and extending to the end of the calling-object string.
  - `public String substring(int beginIndex, int afterEndIndex)`
    - Returns a string that is a subset of the calling-object string, starting at the `beginIndex` position and extending to the `afterEndIndex-1` position.

The `String` Class's `substring` Method

calling object

```
public class StringMethodDemo
{
  public static void main(String[] args)
  {
    String yoda =
      "May the force be with you.";
    System.out.println(yoda.substring(8));
    System.out.println(yoda.substring(4, 13));
  } // end main
} // end StringMethodDemo
```

The `String` Class's `indexOf` Methods

- Here are API headers and brief descriptions for four alternative forms of the `indexOf` method:
  - `public int indexOf(int ch)`
    - Returns the position of the first occurrence of the given `ch` character within the calling-object string. Returns -1 if `ch` is not found.
  - `public int indexOf(int ch, int fromIndex)`
    - Returns the position of the first occurrence of the given `ch` character within the calling-object string, starting the search at the `fromIndex` position. Returns -1 if `ch` is not found.
  - `public int indexOf(String str)`
    - Returns the position of the first occurrence of the given `str` string within the calling-object string. Returns -1 if `str` is not found.
  - `public int indexOf(String str, int fromIndex)`
    - Returns the position of the first occurrence of the given `str` string within the calling-object string, starting the search at the `fromIndex` position. Returns -1 if `str` is not found.

The `String` Class's `indexOf` Methods

```
public class StringMethodDemo
{
  public static void main(String[] args)
  {
    String egyptian =
      "I will not leave the square. Over my dead body.";
    int index = egyptian.indexOf('.');
    String egyptian2 = egyptian.substring(index + 2);
    System.out.println(egyptian2);
  } // end main
} // end StringMethodDemo
```

The `String` Class's `lastIndexOf` Methods

- The `lastIndexOf` methods are identical to the `indexOf` methods except that they search the calling-object string from right to left.
- For the one-parameter `lastIndexOf` method, the search starts from the rightmost character.
- For the two-parameter `lastIndexOf` method, the search starts from the position specified by the second parameter.

- What does this code fragment print?

```
String quote =
  "Peace cannot be kept by force; it can" +
  " only be achieved by understanding."
System.out.print(
  quote.indexOf("can") + " " +
  quote.indexOf("can", 7) + " " +
  quote.lastIndexOf("can"));
```

Formatted Output with the `printf` Method

- You should strive to make program output be understandable and attractive. To further that goal, format your output. Here's an example of formatted output:

```
Account                          Actual      Budget   Remaining
-------                          ------      ------   ---------
Office Supplies                1,150.00    1,400.00      250.00
Photocopying                   2,100.11    2,000.00    (100.11)

Total remaining: $149.89
```

- The `System.out.printf` method is in charge of generating formatted output.

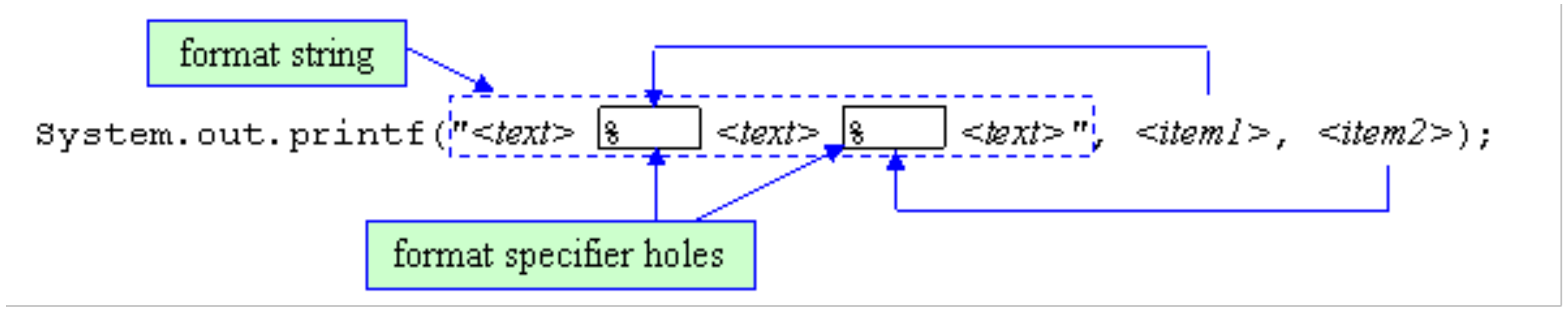
Formatted Output with the `printf` Method

- Here's how to generate the "Total remaining" line in the previous slide's budget report:

format specifier

```
System.out.printf(
  "\nTotal remaining: $%.2f\n", remaining1 + remaining2);
```

- You can have as many format specifiers as you like in a given format string. For each format specifier, you should have a corresponding data item/argument.


format string
System.out.printf("<text> % <text> % <text>", <item1>, <item2>);
format specifier holes

Format Specifier Details

- Here's the syntax for a format specifier:

  `%`[*flags*][*width*][`.`*precision*]*conversion-character*

- The square brackets indicate that something is optional. So the flags, width, and precision parts are optional. Only the % and the conversion character are required.

Conversion Character

- The conversion character tells the Java Virtual Machine (JVM) the type of thing that is to be printed.
- Here is a partial list of conversion characters:

s	This displays a string.
d	This displays a decimal integer (an `int` or a `long`).
f	This displays a floating-point number (a `float` or a `double`) with a decimal point and at least one digit to the left of the decimal point.
e	This displays a floating-point number (`float` or `double`) in scientific notation.

Conversion Character

- This code fragment illustrates how to use the conversion characters:

```
System.out.printf("Planet: %s\n", "Neptune");
System.out.printf("Number of moons: %d\n", 13);
System.out.printf("Orbital period (in earth years): %f\n", 164.79);
System.out.printf(
  "Average distance from the sun (in km): %e\n", 4498252900.0);
```

- Here is the output:

```
Planet: Neptune
Number of moons: 13
Orbital period (in earth years): 164.790000
Average distance from the sun (in km): 4.498253e+09
```

Precision and Width

- Precision:
  - Applies only to floating-point numbers (i.e., it works only in conjunction with the f and e conversion characters).
  - Its syntax consists of a dot and then the number of digits that are to be printed to the right of the decimal point.
  - If the data item has more fractional digits than the precision specifier's value, then rounding occurs. If the data item has fewer fractional digits than the precision specifier's value, then zeros are added at the right so the printed value has the specified number of fractional digits.

- Width:
  - Specifies the minimum number of characters that are to be printed. If the data item contains more than the specified number of characters, then all of the characters are printed. If the data item contains fewer than the specified number of characters, then spaces are added.
  - By default, output values are right aligned, so when spaces are added, they go on the left side.

Precision and Width

- This code fragment illustrates how to specify precision and width in a format specifier:

```
System.out.printf("Cows are %6s\n", "cool");
System.out.printf("But dogs %2s\n", "rule");
System.out.printf("PI = %7.4f\n", Math.PI);
```

- Here is the output:

6 characters

```
Cows are   cool
But dogs rule
PI =  3.1416
```

7 characters

Flags

- Flags allow you to add supplemental formatting features, one flag character for each formatting feature. Here's a partial list of flag characters:

-	Display the printed value using left justification.
0	If a numeric data item contains fewer characters than the width specifier's value, then pad the printed value with leading zeros (i.e., display zeros at the left of the number).
,	Display a numeric data item with locale-specific grouping separators. In the United States, that means commas are inserted between every third digit at the left of the decimal point.
(	Display a negative numeric data item using parentheses, rather than a minus sign. Using parentheses for negative numbers is a common practice in the field of accounting.

BudgetReport Example

left justification

parentheses for negatives, comma for group separators

```
public class BudgetReport
{
  public static void main(String[] args)
  {
    final String HEADING_FMT_STR = "%-25s%13s%13s%15s\n";
    final String DATA_FMT_STR = "%-25s%,13.2f%,13.2f%(,15.2f\n";
    double actual1 = 1149.999; // amount spent on 1st account
    double budget1 = 1400;     // budgeted for 1st account
    double actual2 = 2100.111; // amount spent on 2nd account
    double budget2 = 2000;     // budgeted for 2nd account
    double remaining1, remaining2; // unspent amounts

    System.out.printf(HEADING_FMT_STR,
      "Account", "Actual", "Budget", "Remaining");
    System.out.printf(HEADING_FMT_STR,
      "-------", "------", "------", "---------");
```

BudgetReport Example

```
    remaining1 = budget1 - actual1 ;
    System.out.printf(DATA_FMT_STR,
      "Office Supplies", actual1, budget1, remaining1);
    remaining2 = budget2 - actual2;
    System.out.printf(DATA_FMT_STR,
      "Photocopying", actual2, budget2, remaining2);

    System.out.printf(
      "\nTotal remaining: $%(,.2f\n", remaining1 + remaining2);
  } // end main
} // end class BudgetReport
```