***The Java Lesson 1.***

**What is Java?**

**Hello dear friends!**

With help of [Jon Huhtala](http://javafaq.nu/java_lessons/john_info.htm" \t "_blank) we start today **46 Java Free lessons** course on our site. All the lessons are written by Jon. We just publish his lessons and say: Thank you very much! We lucked the info for beginners until today Smile

Java history

* The fastest growing programming language in the history of computing
* Developed by Sun Microsystems
* Originally designed in 1991 for use in embedded consumer applications
* Redesigned in early 1995 with Internet application capabilities
* Introduced in May, 1995 and immediately supported by the Netscape Web browser
* Rapidly overtaking C++ in popularity among commercial software developers
* Threatened by Microsoft's .NET initiative and C# programming language. The future is uncertain as the legal battle between Microsoft and Sun Microsystems may last for years.

Benefits of Java

* Architecturally neutral. Once written, the same Java program can be run on any platform (computer and operating system) that supports Java.
* Entirely object-oriented. Existing code can be easily re-used and maintained.
* Secure. Dangerous program actions are prohibited.
* Supports Internet programming. Java applets are run by Web browsers.
* Robust. Language features and packaged code support graphical programming, exception handling, networking, multi-threading, I/O, and more.

How Java works

* To develop and distribute a Java program
  1. Programmer codes Java source statements
  2. Java compiler converts source statements into bytecode (platform-independent object language)
  3. Bytecode is copied to the target platform
* To execute a Java program
  1. Platform-dependent Java Virtual Machine (JVM) software must be installed
  2. A copy of the JVM is started
  3. Each bytecode statement is interpreted (translated) by the JVM into platform-dependent machine language and executed under JVM control

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Java source statements | | -> | |  | | --- | | Java compiler | | -> | |  | | --- | | **bytecode** | | ---> | |  |  | | --- | --- | | Platform   |  | | --- | | **Java Virtual Machine** | | |

Disadvantages of Java

* Not supported by all platforms (though third-party JVM software is usually available)
* Slower in execution than compiled languages
* Restricts or prohibits machine-level operations required by certain applications (operating systems, etc.)

Benefits of Java far out-weigh its disadvantages

* Applications require a single set of source files, a single compile, and can be run on any platform having a JVM
* Development time and cost are greatly reduced
* Maintenance time and cost are greatly reduced

The lesson [continues here](http://javafaq.nu/article381.html)

Java programmers

* Are in high demand. For current information, visit any of the following Web sites and use their advanced search facilities to find jobs with "Java" in the job description or requirements. You may find it interesting to compare the popularity of Java with other languages such as C++, COBOL, and VB.

[www.dice.com](http://www.dice.com)

[www.computerjobs.com](http://www.computerjobs.com)

[www.monster.com](http://www.monster.com)

* Are well paid

"A 2001 survey found that U.S. Java developers earn an average of $83,000 per year, making them some of the highest-paid specialists in IT."

- ComputerWorld (May 20, 2002)

* Benefit from certification

"There are hundreds of Java certification options. Recruiters say Sun's offerings carry the most weight, but any certification may help. Only about 25% of Java developers are certified."

- ComputerWorld (May 20, 2002)

* Can easily switch to C++ or C# if necessary. [The languages are very similar](http://www.25hoursaday.com/CsharpVsJava.html).

What does it take to become a certified Java programmer?

* A thorough knowledge of programming logic
* A thorough knowledge of object-oriented programming fundamentals
* Attention to detail
* Java language specifics
* Test-taking skills
* Persistence and practice

**Review questions**

1. Which of the following are true? (choose two)
2. All Java programs are object-oriented.
3. A Java program will execute faster than an equivalent program written in C++.
4. The bytecode for a particular Java program will be the same for different platforms.
5. Java can only be used to create Internet applications (applets).
6. Java was developed by Microsoft.
7. The Java compiler (choose two)
8. reads bytecode
9. writes bytecode
10. is the same regardless of the target platform
11. is also known as the Java Virtual Machine (JVM)
12. translates Java source statements into the machine language of the target platform
13. The Java Virtual Machine
14. is a hardware device
15. is the same on all platforms
16. reads Java source statements
17. is not required to execute a Java program
18. none of the above
19. Which of the following would be the least likely applications to be developed in Java? (choose two)
20. A game for a hand-held, Personal Digital Assistant (PDA)
21. A program to display a map for an on-board car computer
22. A *Windows* program for updating a small disk file
23. A banking application to record up to 500 transactions every second
24. A program to reformat the hard drive of a personal computer

C

***The Java Lesson 2***

**Anatomy of a simple Java program**

Java development software

In order to code and test Java programs, you need Java development software and must know how to use it. Before you go on, you MUST read and master the essential skill of [running Java programs](http://javafaq.nu/nuke_templates/running.htm).

A sample Java program

The following program is a slight variation of the program that is part of the original Test project. When executed, it will display the message "Hello World!" on the system output device (the computer console):

/\*\*  
\* This class defines a program that displays a message on the console.  
\*  
\* Written by: Jon Huhtala  
\*/  
  
public class App {  
  
// This required method is the starting point for processing.  
  
public static void main(String[] args) {  
System.out.println("Hello World!"); // Display the message.  
}  
}

Testing the sample program

Assuming you have a Test project and have already opened a *Notepad* window and a *Command Prompt* window, the procedure to test this sample program is as follows:

1. "Copy" the source statements (shown in blue above).
2. "Paste" the source statements into your *Notepad* window being sure to replace the entire contents of the window if it contained previous code.
3. Save the source file under the name "App.java" in your Test project.
4. Compile the "App.java" source file by entering the following command in the *Command Prompt* window:

javac App.java

1. Run the "App.class" bytecode file by entering the following command in the *Command Prompt*:

java App

Program analysis

Studying this sample program line-by-line will start you on the path to learning Java. The sections that follow will help you analyze the sample program. For now, it is not important that you understand everything completely. Through constant usage, you will eventually know what it all means.

Comments in Java

* Are for documentation purposes only. The compiler ignores them when generating bytecode.
* Three comment techniques are supported

1. Javadoc. Begin with /\*\* and end with \*/. Everything between is treated as a comment.
2. Single-line. Begin with // and automatically continue to the end of the line.
3. Multi-line. Begin with /\* and end with \*/. Everything between is treated as a comment.

* Comments make programs easier to maintain. Good programmers document well enough that a person not familiar with the program can read the comments to know what each section of code is doing.

The lesson [continues here](http://javafaq.nu/article382.html)

The program's class

* Every Java program must be defined as a class. The statement

public class App

is the class header. It declares that a class named App is to be defined and that it is to be publicly available for access from other places. In order for the Java Virtual Machine to find and use the program's class, always make it public. Other access modifiers (private, protected, and default or package access) will be presented later.

* The words public and class are [Java keywords](http://java-help.com/JavaKeywords.htm) and have special meaning. The class name is up to the programmer but must adhere to the restrictions on Java identifiers to be covered in the next lesson.
* The name of the class must be the same as the name of the source file. If the class name is App, it must reside in a file named App.java

Statement blocks

* Are modules of code consisting of one or more statements
* Begin with an opening brace "{" and end with a closing brace "}"
* May be entirely nested within another statement block. For example, the block for the main method is nested within the block for the App class in the sample program.

The main method

* A required method (program module) in all Java programs except applets
* The first method executed by the JVM when processing begins
* In a small program, it defines application processing. In larger programs, it is the highest-level module and calls other methods to perform detailed processing.
* Should have a method header coded as follows:

public static void main(String[] args)

This states that the method has public access, is a class (static) method not to be associated with any objects of the class, does not return a value to the caller (void), has the name main, and receives an array of string references (String[]) named args as parameters. It is not important that you understand all this now, just code the statement exactly as shown. The only flexibility you have in coding the header is the name of the array. The sample program will run just fine if you re-code the method header as

public static void main(String[] x)

Writing to the system output device (the console)

* The system output device is already open and may be referenced by the identifier System.out where out is a public reference to a PrintStream defined within the System class. Again, it is not important that you understand all this now.
* println is a public method of PrintStream objects that accepts a string parameter and copies it to the output device. In the sample program, the string is coded as a string literal enclosed within quotation marks.

Compiler requirements

* All Java statements are delimited by a semicolon ";"
* The compiler ignores "white space" characters (spaces, tabs, and the use of the return key). A good programmer, however, will use "white space" liberally to make their code easier to write, read, and maintain.

DO NOT write code like this

public class App {public static void main(String[] args){  
System.out.println("Hello World!");}}

even though it compiles and runs just fine. Try it and see!

**Review questions**

1. Which of the following are invalid headers for the main method (choose three)
2. public static void Main(String[] args)
3. public static void main(String[] parms)
4. public static void main(String args)
5. public static void main(string[] args)
6. public static void main(String[] args)
7. What will result from attempting to compile and execute the following?

public class App  
{  
public static void main(String[] parms)  
{  
// System.out.println("Java is fun");  
}  
}

1. it will not compile
2. it will compile but an error will occur at run time
3. it will compile and run but nothing will display
4. it will compile and run to display the message "Java is fun" on the console
5. What will result from attempting to compile and execute the following?

public class App {  
public static void main(String[] parms) {  
System.out.println("Java is fun");  
}  
}

1. it will not compile
2. it will compile but an error will occur at run time
3. it will compile and run but nothing will display
4. it will compile and run to display the message "Java is fun" on the console
5. What will result from attempting to compile and execute the following?

public class App  
{  
public static void main(String[] args)  
{  
System.out.println("Java is fun")  
}  
}

1. it will not compile
2. it will compile but an error will occur at run time
3. it will compile and run but nothing will display
4. it will compile and run to display the message "Java is fun" on the console

1. *Uses of identifiers in Java* | 2. [Primitive data types in Java](http://www.javafaq.nu/java-lessons-by-jon-huhtala/primitive-data-types-story.html)

**Identifiers and primitive data types**

**Uses of identifiers**

* To name a variable (a data area that takes on different values during processing)
* To name a method (a module of code)
* To name a class (a definition from which objects can be constructed)
* To label a statement (for later referencing)

**Rules for identifiers**

* It must not be a [Java keyword](http://javafaq.nu/nuke_templates/javakeywords.htm)
* It must begin with a letter, dollar sign ( $ ), or underscore ( \_ )
* Subsequent characters may be letters, dollar signs, underscores, or digits
* Case matters. For example, the three identifiers MAIN, Main, and main are different.

**Examples of valid identifiers**

yearlyGrossPay

\_2002\_Budget

JuneActual

$totalAmtDue

**Programming conventions**

* Are not enforced by the compiler but make it easier to read and maintain Java code
* Most programmers use "camel caps" (such as firstQuarterSalesTotal) to handle long identifiers that appear to contain multiple words. Some use undercores (like first\_Quarter\_Sales\_Totals).
* Begin variable names, method names, and labels with a lower case letter. For example, totalAmount, hoursWorked, employeeName, calcBonus, and main.
* Begin class names with a capital letter. For example, Person, Customer, SalesRep, and Part.
* Names of constants should be all capitals. For example, TAX\_RATE, COMMISSION, and DEPOSIT\_CODE.

***The Java Lesson 4***

**Variables, constants, and literals**

**Variables**

* Data areas that may take on different values during processing
* "Automatic" or "method local" variables are part of a method or nested statement block
* "Member" or "class" variables are part of a class but not part of a method or nested statement block. They will be covered later.

**"Automatic" or "method local" variables**

* Are declared using one of the following general syntax styles:

*data-type identifier*;   
*data-type identifier* = *literal*;   
*data-type identifier* = *expression*;

Example: This program defines three integer variables of different sizes. Two of the variables are assigned values which are displayed by using the println() method.

public class App {  
public static void main(String[] args) {  
byte x;  
short y = 4;  
int z = y + 1;  
System.out.println("y is " + y);  
System.out.println("z is " + z);  
}  
}

Note: When the plus sign (+) is coded following a string, it means concatenation. In each of the println() method calls of this sample, the value of the variable is automatically converted to a string and appended to the end of the string to be displayed.

* Must be initialized before they are used or a compile error will result. For example, the following program will not compile:

public class App {  
public static void main(String[] args) {  
int x;  
int y = x - 6;  
System.out.println("y has a value of " + y);  
}  
}

The expression x - 6 can not be calculated because x has no value.

* Multiple variables of the same type can be declared in a single statement but it doesn't document very well and should be avoided. For example, this program declares four byte variables and displays the values of the two that were initialized:

public class App {  
public static void main(String[] args) {  
byte a, b = -7, c = 5, d;  
System.out.println("b = " + b);  
System.out.println("c = " + c);  
}  
}

Notice how much easier it is to see the declaration of each variable in the following equivalent example:

public class App {  
public static void main(String[] args) {  
byte a;  
byte b = -7;  
byte c = 5;  
byte d;  
System.out.println("b = " + b);  
System.out.println("c = " + c);  
}  
}

* Can be referenced only in the block in which they are defined and blocks nested within that block ("inner" blocks). This is known as the "scope" of the variable. For example, the program

public class App {  
public static void main(String[] args) {  
int x = 17;  
{  
System.out.println("x = " + x);  
}  
}  
}

will compile and run because variable x is known within the inner block. The following program, however, will not compile:

public class App {  
public static void main(String[] args) {  
{  
int x = 17;  
}  
System.out.println("x = " + x);  
}  
}

The variable x is not "in scope" in the outer block.

* Are automatically deleted ("garbage collected") when processing jumps to an outer block of code. The memory space they occupied is made available for other uses. In the previous program, the variable x would no longer exist when the println() method is to be called. That is the reason for the compile error.

**Literals**

* Are constants having no identifier
* Have their value specified within the program's source code
* Can only appear on the right side of an assignment operator ( = ) or within an expression
* Have a data type associated with them

**boolean literals**

* Can only have the value true or false
* Can only be assigned to booleanvariables

Example: This program declares and initializes two boolean variables, then displays their values.

public class App {  
public static void main(String[] args) {  
boolean isToday = true;  
boolean isTomorrow = false;  
System.out.println("Is it today? " + isToday);  
System.out.println("Is it tomorrow? " + isTomorrow);  
}  
}

**char literals**

* Represent a single Unicode character (16 bits)
* Must be enclosed within single quotes (apostrophes)
* Are often associated with a single key stroke
* Can represent special characters ("escape sequences") used for device control

Example: This program declares a number of char variables then displays the values of a few of them.

public class App {  
public static void main(String[] args) {  
  
// Some char literals for keys found on the standard keyboard  
  
char lowerCaseA = 'a';  
char upperCaseA = 'A';  
char digit3 = '3';  
char space = ' ';  
  
// "Escape" sequences for denoting special characters  
  
char newLine = ' ';  
char returnKey = ' ';  
char tab = ' ';  
char backspace = '';  
char formfeed = 'f';  
char singleQuote = ''';  
char doubleQuote = '"';  
char backslash = '';  
  
// Hexadecimal, Unicode value for the Yen currency symbol  
  
char yenSymbol = 'u00a5';  
  
// Display the values of some of the variables declared above.  
  
System.out.println("Selected values: " + digit3 + newLine + tab +  
backslash + space + yenSymbol);  
}  
}

**Integer literals**

* Represent an integer value
* Can be expressed in decimal (the default), octal (base Cool, or hexadecimal (base 16)
* Are not enclosed in any special characters
* Are automatically int (32 bits) unless the suffix 'L' is appended to make it long (64 bits)

Example: This program declares and initializes several integer variables and displays some of their values.

public class App {  
public static void main(String[] args) {   
  
// The decimal value 28 expressed as a decimal literal  
  
byte x = 28;  
  
// The decimal value 28 expressed as an octal literal. The value  
// must begin with "0" and the digits must be in the range 0 to 7.  
  
byte xAsOctal = 034;  
  
// The decimal value 28 expressed as a hexadecimal literal. The value  
// must begin with "0x" and the digits must be in the range 0 to F.  
// Upper and lower case letters are accepted.  
  
byte xAsHex\_1 = 0x1c;  
byte xAsHex\_2 = 0x1C;  
byte xAsHex\_3 = 0X1c;  
byte xAsHex\_4 = 0X1C;  
  
// Decimal value 123456789 as a long literal  
  
long bigOldUselessNumber = 123456789L;   
  
// Display some of the values  
  
System.out.println(x);  
System.out.println(xAsOctal);  
System.out.println(xAsHex\_3);  
System.out.println(bigOldUselessNumber);  
}  
}

**Floating-point literals**

* Represent a real number (having a decimal point)
* Can be expressed as a standard decimal value or in scientific notation
* Are not enclosed in any special characters
* Are automatically double (64 bits) unless the suffix 'F' is appended to make it float (32 bits)

Example: This program declares and initializes several floating-point variables and displays some of their values.

public class App {  
public static void main(String[] args) {  
  
// Some double literals in both standard and scientific notation  
  
double amount = 1234.56;  
double amountInScientificNotation = 1.23456e+3;  
double tiny = .0000123;  
double tinyInScientificNotation = 1.23e-5;  
  
// For a literal to be float, "F" or "f" must be appended  
  
float value = 567.89F;  
float valueInScientificNotation = 5.6789e+2F;  
  
// Display some of the values  
  
System.out.println(amount);  
System.out.println(amountInScientificNotation);  
System.out.println(tiny);  
System.out.println(tinyInScientificNotation);  
System.out.println(value);  
}  
}

**String literals**

* Represent a string of characters, such as "Java is fun"
* Must be enclosed in double quotes
* Are automatically stored as String class objects by the compiler. They will be covered later.

Example: This program contains three string literals (look for the double quotes).

public class App {  
public static void main(String[] args) {  
  
// Declare a String object and initialize it.  
  
String firstLine = "At Ferris State University,";  
  
// Display the string after advancing to a new line and tabbing.  
// Then, display another string  
  
System.out.println(" " + firstLine);  
System.out.println(" Java is fun!");  
}  
}

**Constants**

* Are similar to variables but, once initialized, their contents may NOT be changed
* Are declared with the keyword final
* By convention, have all capital letters in their identifier. This makes them easier to see within the code.

Example 1: This program defines a number of constants and then displays some of their values.

public class App {  
public static void main(String[] args) {  
  
final boolean YES = true;  
final char DEPOSIT\_CODE = 'D';  
final byte INCHES\_PER\_FOOT = 12;  
final int FEET\_PER\_MILE = 5280;  
final float PI = 3.14F;  
final double SALES\_TAX\_RATE = .06;  
final String ADDRESS = "119 South Street";  
  
// Display some of the values  
  
System.out.println(INCHES\_PER\_FOOT);  
System.out.println(ADDRESS);  
}  
}

Example 2: This program will not compile because an attempt is made to change the value of its constant.

public class App {  
public static void main(String[] args) {  
  
final double SALES\_TAX\_RATE = .06;  
SALES\_TAX\_RATE = .04;  
  
// Display the sales tax rate  
  
System.out.println(SALES\_TAX\_RATE);  
}  
}

***The Java Lesson 5***

**Arithmetic operations, conversions, and casts**

**Basic arithmetic operations**

* Involve two "operands" that must be numeric primitive values (non-boolean). Because char data has a positive integer equivalent, it may be used in arithmetic operations.
* Appear in expressions. The calculated result of the expression is often assigned to a variable according to the following general syntax:

*identifier* = *expression*;

* Five arithmetic operations are part of the Java language. Other useful arithmetic operations, such as exponentiation and trigonometric functions, are provided as packaged code in the Math class and will be covered later.

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| + | Addition |
| - | Subtraction |
| \* | Multiplication |
| / | Division |
| % | Modulo (remainder) |

**Addition, subtraction, and multiplication**

* Act as you would expect. They calculate the sum, difference, and product of two numeric operands.

**Example**: This program reads two integer values from the user. It then displays the sum, difference, and product of the two numbers.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two integer values entered by the user  
  
int x;  
int y;  
  
// Prompt for and read the two integer values  
  
System.out.print("First integer: ");  
x = Keyboard.readInt();  
System.out.print("Second integer: ");  
y = Keyboard.readInt();  
  
// Display the sum, difference, and product of the two integers  
  
System.out.println(" " + x + " + " + y + " = " + (x + y));  
System.out.println(" " + x + " - " + y + " = " + (x - y));  
System.out.println(" " + x + " \* " + y + " = " + (x \* y));  
}  
}

Notes:

1. This sample reads data from the keyboard, which isn't easy in Java. My custom Keyboard class makes it easier. For information on using the class, [click here](http://java-help.com/Keyboard.htm).
2. The prompt messages use the print() method instead of println(). They are virtually the same except that print() leaves the insert point at the end of the current line so that the user's entry will appear on the same line as the prompt.

* Overflow or underflow will result in an erroneous value if the calculation is mathematically too large or too small to be properly stored in memory

**Example**: This program results in an overflow and displays an erroneous sum.

public class App {  
public static void main(String[] args) {  
  
// Two extremely large integer values  
  
int x = 2000000000; // 2 billion.  
int y = 1000000000; // 1 billion.  
  
// Display an erroneous sum due to overflow  
  
System.out.println("The erroneous sum is " + (x + y));  
}  
}

**Division**

* When both operands are integers, the result is the integer quotient obtained by dividing the first operand (the dividend) by the second operand (the divisor)
* The sign of the result is determined by the rules of algebra
* When both operands are integers and the second operand is zero, a run time exception occurs

Example: The following program reads two integer values from the user and displays the result of dividing the first number by the second number. If the user enters 0 for the second number, a runtime exception will occur.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two integer values entered by the user  
  
int x;  
int y;  
  
// Prompt for and read the two integer values  
  
System.out.print("First integer (dividend): ");  
x = Keyboard.readInt();  
System.out.print("Second integer (divisor): ");  
y = Keyboard.readInt();  
  
// Calculate and display the integer quotient  
  
System.out.println(" " + x + " / " + y + " = " + (x / y));  
}  
}

Note: When testing this code, be sure to try division by zero.

* When either operand is floating-point, a floating-point value will result that is an approximation. For example, the following program will give you a result other than the expected "3.0"

public class App {  
public static void main(String[] args) {  
System.out.println(2.1/.7);  
}  
}

* When either number is floating-point, division by zero is allowed. The result will be + infinity.

Example: This program reads two floating-point values from the user. It then displays the result of dividing the first number by the second number. If the user enters 0 for the second number, a result of + infinity will occur.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two floating-point values entered by the user  
  
double x;  
double y;  
  
// Prompt for and read the two floating-point values  
  
System.out.print("First floating-point number (dividend): ");  
x = Keyboard.readDouble();  
System.out.print("Second floating-point number (divisor): ");  
y = Keyboard.readDouble();  
  
// Calculate and display the result  
  
System.out.println(" " + x + " / " + y + " = " + (x / y));  
}  
}

Note: When testing this code, be sure to try division by zero.

**Modulo (remainder)**

* When both operands are integers, the result is the integer remainder obtained by dividing the first operand by the second operand.
* The sign of the result is the same as the sign of the first operand.
* When both operands are integers and the second operand is zero, a run time exception occurs

Example: This program reads two integer values from the user. It then displays the remainder that results from dividing the first number by the second number. If the user enters 0 for the second number, a runtime exception will occur.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two integer values entered by the user  
  
int x;  
int y;  
  
// Prompt for and read the two integer values  
  
System.out.print("First integer (dividend): ");  
x = Keyboard.readInt();  
System.out.print("Second integer (divisor): ");  
y = Keyboard.readInt();  
  
// Calculate and display the integer remainder  
  
System.out.println(" " + x + " modulo " + y + " = " + (x % y));  
}  
}

Note: When testing this code, try different integer values with different signs. Also try a second operand of zero.

* When either number is floating-point, the result is a floating-point approximation of the fractional value remaining after an integral number of subtractions. For example,

2.2 % .7 results in 0.10000000000000031

This is obtained by subtracting .7 from 2.2 until the result is less than .7

* The sign of the result is the same as the sign of the first operand
* When either number is floating-point, division by zero is allowed. The result will be NaN (meaning Not a Number).

Example: This program reads two floating-point values from the user. It then displays the remainder that results from dividing the first number by the second number.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two floating-point values entered by the user  
  
double x;  
double y;  
  
// Prompt for and read the two floating-point values  
  
System.out.print("First floating-point number (dividend): ");  
x = Keyboard.readDouble();  
System.out.print("Second floating-point number (divisor): ");  
y = Keyboard.readDouble();  
  
// Calculate and display the floating-point remainder  
  
System.out.println(" " + x + " modulo " + y + " = " + (x % y));  
}  
}

Note: When testing this code, try different floating-point values with different signs. Also try a second operand of zero.

**Arithmetic assignment operations**

* Combine arithmetic and assignment into a single operation to save coding time
* All five basic arithmetic operations have a corresponding arithmetic assignment operator

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| += | Addition assignment |
| -= | Subtraction assignment |
| \*= | Multiplication assignment |
| /= | Division assignment |
| %= | Modulo (remainder) assignment |

Example: This program reads a percentage value from the user (such as 5.5) and converts to the correct decimal equivalent (such as 0.055) by using the division assignment operator.

public class App {  
public static void main(String[] args) {  
  
// Variable for holding a percentage entered by the user  
  
double percent;  
  
// Prompt for and read the percentage  
  
System.out.print("Enter a percentage in n.nn format: ");  
percent = Keyboard.readDouble();  
  
// Convert it to its decimal equivalent and display the result  
  
percent /= 100;  
System.out.println("The decimal equivalent is " + percent);  
}  
}

Note: The arithmetic assignment operators always result in the first operand (the one on the left) being modified.

**Increment and decrement operations**

* Require only one operand. They are said to be "unary" operators.
* Are the most efficient way to add or subtract one from a numeric variable

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| ++ | Increment (add one) |
| -- | Decrement (subtract one) |

* Can be placed on the left or right of a numeric variable. On the left, the increment or decrement occurs *before* the expression is evaluated. On the right, the increment or decrement occurs *after* the expression has been evaluated.

Example: The following program reads two integer values from the user. It increments the first value and decrements the second.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two integer values to be entered by the user  
  
int x;  
int y;  
  
// Prompt for and read the two values  
  
System.out.print("Enter an integer to be incremented: ");  
x = Keyboard.readInt();  
System.out.print("Enter an integer to be decremented: ");  
y = Keyboard.readInt();  
  
// Perform the increment and decrement and display the results  
  
x++;  
System.out.println("The new values are " + x + " and " + --y);  
}  
}

Note: It is best to use a stand-alone statement when using the ++ or -- operators. It makes your code easier to read.

**Conversions**

* Are performed automatically when mixed numeric types appear within an expression. Variables and constants are not altered, but their values are copied to intermediate areas of larger size or precision to prevent possible data loss during the calculation. This is known as a "widening" conversion.

The order of conversion (from narrowest to widest) is as follows. **Memorize this table!**

|  |  |  |
| --- | --- | --- |
| byte |  |  |
| | |  |  |
| short |  | char |
|  |  | / |
|  | int |  |
|  | | |  |
|  | long |  |
|  | | |  |
|  | float |  |
|  | | |  |
|  | double |  |

* The data type resulting from an expression is that of the largest or most precise variable or constant appearing in the expression but is never smaller than int. Programmers that fail to understand this are constantly plagued by compile errors.

Example: This program to convert a fahrenheit temperature to celsius will NOT compile.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding fahrenheit and celsius temperatures  
  
double tempF;  
float tempC;  
  
// Prompt for and read the fahrenheit temperature  
  
System.out.print("Enter the fahrenheit temperature (nn.n): ");  
tempF = Keyboard.readDouble();  
  
// Convert to celsius and display the result  
  
tempC = 5 \* (tempF - 32) / 9;  
System.out.println("Celsius equivalent is " + tempC);  
}  
}

Note: To fix the compile error in this code, tempC must be double or tempF must be float. Alternatively, the result of the expression can be "cast" as float (the next topic).

**Casts**

* Are a programmer's way of telling the compiler to ignore possible loss of data or precision that might result from a conversion
* Should be used with caution. Data loss is not to be taken lightly. The general syntax is

*identifier* =(*data-type*) *expression*;

Example: This is a corrected version of the previous program that uses a cast to prevent the compile error.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding fahrenheit and celsius temperatures  
  
double tempF;  
float tempC;  
  
// Prompt for and read the fahrenheit temperature  
  
System.out.print("Enter the fahrenheit temperature (nn.n): ");  
tempF = Keyboard.readDouble();  
  
// Convert to celsius and display the result  
  
tempC = **(float)** **(**5 \* (tempF - 32) / 9**)**;  
System.out.println("Celsius equivalent is " + tempC);  
}  
}

Note: The result of the entire expression is double but is cast as float in order to be assigned to the float variable. The additional set of parenthesis are coded around the entire expression to prevent only the integer literal 5 from being cast.

**Some thoughts on arithmetic expressions**

Arithmetic expressions can be large and complex with several variables, constants, and operators. The order in which operations are performed is the same as in mathematics (multiplication and division first, then addition and subtraction, working from left to right). To avoid confusion, good programmers keep their expressions as simple as possible and use parenthesis for clarity.

***The Java Lesson 6***

**Boolean expressions and operations**

**Boolean expressions**

* Always result in a boolean value (either true or false)
* Are used to resolve logic questions, such as determining if two variables have the same value
* May contain several different operators

**Comparison operators**

* Used to compare the values of two variables or constants
* Require that the two operands be either both numeric (including char) or both boolean. A numeric value cannot be compared to a boolean value.
* Frequently appear in if, for, while, and do-while statements (to be covered later)

|  |  |
| --- | --- |
| **Operator** | **Operation** |
| < | Less than |
| <= | Less than or equal |
| > | Greater than |
| >= | Greater than or equal |
| == | Equal |
| != | Not equal |

Note: Java makes a clear distinction between assignment ( = ) and a test for equality ( == ). Failure to understand this leads to many programming errors!

Example: This program reads two numeric values from the user and displays the result of several comparisons involving the two numbers.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding two numeric values entered by the user  
  
int x;  
double y;  
  
// Prompt for and read the two numeric values  
  
System.out.print("First number (integer): ");  
x = Keyboard.readInt();  
System.out.print("Second number (floating-point): ");  
y = Keyboard.readDouble();  
  
// Display information comparing the two values  
  
System.out.println(" " + x + " < " + y + " is " + (x "<" y));  
System.out.println(" " + x + " <= " + y + " is " + (x <= y));  
System.out.println(" " + x + " > " + y + " is " + (x > y));  
System.out.println(" " + x + " >= " + y + " is " + (x >= y));  
System.out.println(" " + x + " == " + y + " is " + (x == y));  
System.out.println(" " + x + " != " + y + " is " + (x != y));  
}  
}

Note: When you test this code, use several different values - including the same value for both numbers.

**Logical operators**

* Combine the results of two Boolean expressions into a single boolean value
* Provide for complex logic. The following table shows the operators and how they can be used to combine the results of two Boolean expressions *X* and *Y*:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Operation** | **Resulting value** true **if...** | ***X* and *Y* always evaluated?** |
| & | AND | *X* and *Y* are both true | Yes |
| | | OR | either *X* or *Y* is true | Yes |
| ^ | XOR (exclusive OR) | *X* and *Y* have different values | Yes |
| && | Conditional AND | *X* and *Y* are both true | No |
| || | Conditional OR | either *X* or *Y* is true | No |

The conditional AND ( && ) and OR ( || ) are sometimes called "short-circuit" operators because the second operand is not always evaluated depending on the value of the first operand. If the first operand is false, the result of an AND will always be false regardless of the value of the second operand. If the first operand is true, the result of an OR will always be true regardless of the value of the second operand.

Example: The following program uses two values entered by the user to determine if a customer is entitled to free shipping. A customer receives free shipping if their order amount is greater than or equal to 100 and they are a preferred customer.

public class App {  
public static void main(String[] args) {  
  
// Variables for holding order amount and valued customer data to be  
// entered by the user  
  
double amount;  
char valuedCust;  
  
// Variables for holding information about free shipping  
  
boolean isFree;  
  
// Prompt for and read order amount and valued customer data  
  
System.out.print("Order amount (floating-point): ");  
amount = Keyboard.readDouble();  
System.out.print("Valued customer? (Y)es or (N)o: ");  
valuedCust = Keyboard.readChar();  
  
// Determine and display information about free shipping.  
// Shipping is free if the order amount is greater than or equal $100  
// AND the customer is a valued customer.  
  
isFree = (amount >= 100 && (valuedCust == 'Y' || valuedCust == 'y'));  
System.out.println(" Free shipping? " + isFree);  
}  
}

Note: When you test this code, use several different values - including upper and lower case 'Y' or 'N' in response to the "Valued Customer?" prompt.

**A warning about comparing strings**

* Strings are objects. If you attempt to compare them using the comparison operators, you are comparing where they are stored in memory (their addresses) and NOT their contents.
* To properly compare the contents of two String objects you should use the equals() method. This will be covered in a later lesson.

Example: This program reads a string from the user and erroneously tests it to determine if they entered "Monday". Because the user's String object and the literal's String object are in different memory locations, the result of the comparison will always be false, even if the user enters "Monday".

public class App {  
public static void main(String[] args) {  
  
// String variable for the day of the week  
  
String weekDay;  
  
// Variable for indicating if the day is Monday  
  
boolean isMonday;  
  
// Prompt for and read the day of the week  
  
System.out.print("Day of the week (Monday, Tuesday, etc.): ");  
weekDay = Keyboard.readString();  
  
// Determine and display information about the day of the week.  
  
isMonday = (weekDay == "Monday");  
System.out.println(" Is the day Monday? " + isMonday);  
}  
}

**Certain logical operators can be combined with assignment**

* The operators are &= (AND equals), |= (OR equals), and ^= (XOR equals)
* The left operand must be boolean

Example: The following program reads a boolean value from the user and displays its opposite value.

public class App {  
public static void main(String[] args) {  
  
// Variable for holding a boolean value entered by the user  
  
boolean value;  
  
// Prompt for and read the boolean value  
  
System.out.print("Enter a boolean value (true or false): ");  
value = Keyboard.readBoolean();  
  
// Determine the opposite value. If the value is false, XOR with  
// true results in true. If the value is true, XOR with true results  
// in false.  
  
value ^= true;  
  
// Display the new value of the value.  
  
System.out.println(" The opposite value is " + value);  
}  
}

Note: An alternate way to set the opposite value is to code.

value = !value;

***The Java Lesson 7***

**Bitwise operations: overview with detailed examples**

1. *Logical bitwise operations*  
2. [The complement operator](http://www.javafaq.nu/java-lessons-by-jon-huhtala/bitwise-operations-complement-operator-story.html)   
3. [Bitwise shift operations](http://www.javafaq.nu/java-lessons-by-jon-huhtala/bitwise-shift-operations-story.html)   
4. [Bitwise assignment operations andr review questions](http://www.javafaq.nu/java-lessons-by-jon-huhtala/bitwise-assignment-operations-story.html)

**Overview**

Bitwise operations act upon individual bits within integer data. They are used to perform the logical operations AND, OR, and XOR (eXclusive OR), complementing (reversing all bits), and shifting (sliding bits to the left or right).

**Logical bitwise operations**

* Use the standard boolean operators (&, |, and ^) to act upon two integer values. The short-circuit boolean operators (&& and ||) are not used for bitwise operations and will result in a compile error if attempted.
* Produce an integer result (of size int or larger) that is the logical AND, OR, or XOR of two operands. The rules for these operations are as follows:

|  |  |
| --- | --- |
| **Operation** | **Rule** |
| & | If both corresponding operand bits are "on" the result bit is "on" |
| | | If either corresponding operand bit is "on" the result bit is "on" |
| ^ | If corresponding operand bits are different the result bit is "on" |

For example, if

byte x = 5; // Binary value: 0000 0101 Hex value: 05  
byte y = 9; // Binary value: 0000 1001 Hex value: 09  
byte z;

this table shows the result of executing three unrelated statements. The cast is needed in order to store the int value that results from the operation.

|  |  |  |  |
| --- | --- | --- | --- |
| **Statement** | **Binary Result** | **Hexadecimal** | **Decimal** |
| z = (byte)(x & y); | 0000 0001 | 01 | 01 |
| z = (byte)(x | y); | 0000 1101 | 0D | 13 |
| z = (byte)(x ^ y); | 0000 1100 | 0C | 12 |

Example: The following program can be run to test logical bitwise operations.

public class App {  
public static void main(String[] args) {  
  
// Variables to be read from the user  
  
int first;  
int second;  
  
// Prompt for and read the two integers  
  
System.out.print("First integer: ");  
first = Keyboard.readInt();  
System.out.print("Second integer: ");  
second = Keyboard.readInt();  
  
// Display the results of logical bitwise operations  
  
System.out.println(" " + first + " & " + second + " = " +  
(first & second));  
System.out.println(" " + first + " | " + second + " = " +  
(first | second));  
System.out.println(" " + first + " ^ " + second + " = " +  
(first ^ second));  
}  
}

Notes:

1. Program results are displayed in decimal. To really understand what is happening, work out the equivalent binary and hexadecimal values.
2. Be sure to run the program several times with different integer values.

***The Java Lesson 8***

**Flow control with if and else**

**Overview**

The if and else statements allow a block of code or a single statement to be either executed or bypassed depending upon the result of a comparison operation. They are an essential part of Java "flow control" by which execution may proceed along alternate logic paths within a program.

**The if statement**

* Identifies a single statement or a block of statements to be executed if an expression evaluates to true
* Has two forms. The general syntax is either of the following:

if (*expression*) *statement*;

if (*expression*) { *statements*;}

The first form is often referred to as a "single statement if". For example,

if (amountDue > 0)  
System.out.println("You owe " + amountDue);

will display the message only if the value of amountDue is greater than zero. The logic path merges at the next statement in the program.

The second form is often referred to as a "multiple statement if". For example,

if (amountDue > 0) {  
totalDueFromAllCustomers += amountDue;  
System.out.println("You owe " + amountDue);  
}

will add to the grand total and display the message only if the value of amountDue is greater than zero. The logic path merges at the first statement after the end of the statement block.

* Can lure unsuspecting programmers into some nasty errors. For example,

if (amountDue > 0);  
System.out.println("You owe " + amountDue);

will display the message no matter what value amountDue has. The accidental semicolon after the comparison expression is to blame. The code says that if the expression is true you want to do nothing. Logic then merges at the next statement, so the message is always displayed.

Another common error is to forget the braces in what was intended to be a multiple statement if. For example,

if (amountDue > 0)  
totalDueFromAllCustomers += amountDue;  
System.out.println("You owe " + amountDue);

will add to the grand total if the value of amountDue is greater than zero. The message, however, will always be displayed. Omitting the braces makes this a single statement if, so the logic path merges at the statement that displays the message.

**The else statement**

* Defines an alternative and mutually exclusive logic path to be followed if the preceding if statement expression evaluates to false
* Must always be paired with, and follow, an if statement. It can never be coded separately.
* Has two forms. The general syntax is either of the following:

else *statement*;

else { *statements*;}

The first form is often referred to as a "single statement else". For example,

if (amountDue > 0)  
System.out.println("You owe " + amountDue);  
else  
System.out.println("You owe nothing");

will display how much is owed if amountDue is greater than zero. Otherwise, the "You owe nothing" message will be displayed. The two logic paths are mutually exclusive and merge at the next statement in the program.

The second form is often referred to as a "multiple statement else". For example,

if (amountDue > 0) {  
totalDueFromAllCustomers += amountDue;  
System.out.println("You owe " + amountDue);  
}  
else {  
customerRating = 'A';  
System.out.println("You owe nothing");  
}

will add to the grand total and display how much is owed if amountDue is greater than zero. Otherwise, the customerRating will be set to 'A' and the "You owe nothing" message will be displayed. The two logic paths are mutually exclusive and merge at the first statement after the end of the else block.

* Can trap programmers who get sloppy. For example,

if (age < 65)  
System.out.println("Regular admission");  
else;  
System.out.println("Senior admission");

will always display the "Senior admission" message no matter what value age has. The accidental semicolon after the else says that if the expression is false you want to do nothing. Logic then merges at the next statement, so the message is always displayed.

**Nesting**

* Is permitted. An if or an if-else can be coded within an if or an if-else.

Example:

public class App {  
public static void main(String[] args) {  
  
int age;  
  
System.out.print("Enter age: ");  
age = Keyboard.readInt();  
  
if (age >= 1Cool {  
if (age < 65) {  
System.out.println("$5 admission");  
}  
else {  
System.out.println("$4 admission");  
}  
}  
else {  
if (age < 5) {  
System.out.println("Free admission");  
}  
else {  
System.out.println("$2 admission");  
}  
}  
  
}  
}

Notes:

1. The program prompts for and reads a person's age from the user.
2. Based upon the value of age, it displays a different admission fee (under 5 is free, 5-17 is 2 dollars, 18-64 is 5 dollars, and 65 and over is 4 dollars).

* Can lead to nasty logic errors. The following code, for example, doesn't do what you initially expect.

if (age < 65)  
if (age < 1Cool {  
System.out.print("Child discount");  
}  
else {  
System.out.print("Senior discount");  
}

It *appears* that for those 65 and over the "Senior discount" message will display, for those under 18 the "Child discount" message will display, and for all others no message will display.

What *actually* displays is nothing for those 65 and over, "Child discount" for those under 18, and "Senior discount" for those from 18-64!!

Because an else is always paired with the last if that didn't have an else and to which it can legally belong, the preceding code is actually the same as the following:

if (age < 65)  
if (age < 1Cool {  
System.out.print("Child discount");  
}  
else {  
System.out.print("Senior discount");  
}

This error can easily be avoided by ALWAYS using the multiple statement syntax for the if and else. For example, the inclusion of two braces in the original code fixes the error:

if (age < 65) **{**  
if (age < 1Cool {  
System.out.print("Child discount");  
}  
**}**  
else {  
System.out.print("Senior discount");  
}

**Example**

The following program demonstrates nested if-else logic. It determines a customer's new balance based upon their starting balance, a transaction amount, and a transaction code entered by the user. A "charge" transaction will add to the customer's balance while "payment" and "return" transactions will subtract from it.

public class App {  
public static void main(String[] args) {  
  
// Variables  
  
double balance;  
char code;  
double amount;  
boolean isGood;  
  
// Prompt for and read data  
  
System.out.print("Enter customer's starting balance: ");  
balance = Keyboard.readDouble();  
System.out.print("Enter transaction amount: ");  
amount = Keyboard.readDouble();  
System.out.println("Transaction codes are");  
System.out.println(" " + "C - charge");  
System.out.println(" " + "P - payment");  
System.out.println(" " + "R - refund or return");  
System.out.print("Enter transaction code: ");  
code = Keyboard.readChar();  
  
// Process based upon transaction code  
  
if (code == 'C' || code == 'c') {  
balance += amount;  
isGood = true;  
}  
else {  
if (code == 'P' || code == 'p') {  
balance -= amount;  
isGood = true;  
}  
else {  
if (code == 'R' || code == 'r') {  
balance -= amount;  
isGood = true;  
}  
else {  
isGood = false;  
}  
}  
}  
  
// Display result  
  
if (isGood) {  
System.out.println("New balance is " + Utility.moneyFormat(balance));  
}  
else {  
System.out.println("Invalid transaction code");  
}  
  
}  
}  
Notes:

1. The balance, code, and amount variables hold data entered by the user. The isGood variable is set during processing to indicate whether the transaction code is valid.
2. After all data has been read from the user, the transaction is processed based upon its transaction code. The comparison expressions allow for the possibility that the user may have entered the transaction code in either upper or lower case.
3. If the transaction code is valid, the balance is adjusted accordingly and isGood set to true (indicating a good transaction). If an invalid transaction code was entered, logic flows to the point where isGood is set to false (indicating a bad transaction).
4. After the transaction has been processed, the result is displayed based upon the value of isGood. For a successful transaction, the customer's new balance is displayed using the moneyFormat() method of my Utility class (for more information about this class and its methods, [click here](http://java-help.com/Utility.htm)). If the transaction was bad, an error message is displayed.
5. Many Java programmers structure nested if-else code to produce what appears to be an else-if. For example, the above statements that process the transaction might be coded as follows:

if (code == 'C' || code == 'c') {  
balance += amount;  
isGood = true;  
}  
else if (code == 'P' || code == 'p') {  
balance -= amount;  
isGood = true;  
}  
else if (code == 'R' || code == 'r') {  
balance -= amount;  
isGood = true;  
}  
else {  
isGood = false;  
}

If you try this code, you will see that it works. An easier way to simplify the code will be presented in the next lesson.

***The Java Lesson 9***

**switch statements**

http://javafaq.nu/dir.php?op=News&edit=412

**Overview**

Flow control with if and else statements gets cumbersome when a variable must be tested for a large number of possible values, such as a menu selection that permits the user to enter an integer from 1 to 20. The solution to this problem is the switch statement.

**The switch statement**

* Specifies an expression whose value is be tested. This is known as the "switch expression" and is typically the name of a single variable.
* Defines a number of "cases", each associated with a different value. There may also be a "default" case that is not associated with a value. If the value of the switch expression is equal to the value of a case, the statements for that case will be executed. Otherwise, the statements of the default case will be executed.
* General syntax:

switch (*expression*) {case *value1*:*statements*;break;case *value2*:*statements*;break;default:*statements*;break;}

* Has a number subtleties and restrictions:

1. The switch expression must be a primitive of type int or able to be promoted to int. Specifically, it must be either byte, short, int, or char. Expressions of type boolean, long, float, and double will not compile.
2. The value specified on a case must be a constant of type int or must be able to be promoted to int (in other words a byte, short, int, or char). It must also be within the possible range of values of the switch expression. For example, if the switch variable is a byte, a case with a value of 200 would not compile because a byte may only have a value from -128 to 127.

The value of a case may be an expression as long as the result is a constant. For example,

case 5 + 1:

would compile successfully.

1. The break statement is optional and will be covered in more detail in a later lesson. When encountered, it ends the execution of the switch statement. If omitted from a case, processing falls through to the next case (a sometimes undesirable result).
2. Although the compiler doesn't care, cases should be arranged in a high probability to low probability order to enhance processing efficiency. The default can be placed anywhere (even first).

* May be nested. A switch can be coded within another switch or in either leg of an if-else. It can also contain if-else code.

**Example**

The following program is a revised version of the nested if-else sample from the previous lesson:

public class App {  
public static void main(String[] args) {  
  
// Variables  
  
double balance;  
char code;  
double amount;  
  
// Prompt for and read data  
  
System.out.print("Enter customer's starting balance: ");  
balance = Keyboard.readDouble();  
System.out.print("Enter transaction amount: ");  
amount = Keyboard.readDouble();  
System.out.println("Transaction codes are");  
System.out.println(" " + "C - charge");  
System.out.println(" " + "P - payment");  
System.out.println(" " + "R - refund or return");  
System.out.print("Enter transaction code: ");  
code = Keyboard.readChar();  
  
// Process based upon transaction code  
  
switch (code) {  
case 'C':  
case 'c':  
balance += amount;  
System.out.println("New balance is " + Utility.moneyFormat(balance));  
break;  
case 'P':  
case 'p':  
balance -= amount;  
System.out.println("New balance is " + Utility.moneyFormat(balance));  
break;  
case 'R':  
case 'r':  
balance -= amount;  
System.out.println("New balance is " + Utility.moneyFormat(balance));  
break;  
default:  
System.out.println("Invalid transaction code");  
break;  
}  
}  
}

Notes:

1. The balance, code, and amount variables hold data entered by the user.
2. After all data has been read from the user, the transaction is processed by the switch statement with the transaction code (code) used as the switch expression.
3. Within the switch, if the value of code matches the value of a particular case, processing jumps to the statement block for that case, otherwise processing jumps to the statement block for the default. Processing will then continue until either a break statement is encountered or the end of the switch is reached.
4. Stacking two or more cases without at break statement constitutes an OR. For example,

case 'R':  
case 'r':

will result in the same processing being performed if the value of code is either an uppercase or lowercase 'R'.

1. For transactions with a valid transaction code, the customer's new balance is displayed using the moneyFormat() method of my Utility class. If the transaction code is bad, an error message is displayed.