Fundamentals of Java
Lesson 3: Syntax, Errors, and Debugging

Text by: Lambert and Osborne

Slides by: Cestroni

Modifications by: Mr. Dave Clausen

Updated for Java 5 (version 1.5)

Lesson 3: Syntax, Errors, and Debugging

Objectives:

- Construct and use numeric and string literals.
- Name and use variables and constants.
- Create arithmetic expressions.
- Understand the precedence of different arithmetic operators.
- Concatenate two strings or a number and a string.
- Know how and when to use comments in a program.
- Tell the difference between syntax errors, run-time errors, and logic errors.
- Insert output statements to debug a program.

Lesson 3: Syntax, Errors, and Debugging

Vocabulary:

- arithmetic expression
- comments
- exception
- literal
- logic error
- package
- pseudocode
- reserved words
- run-time error
- semantics
- syntax
- virus

3.1 Language Elements

- Language elements:
  - Vocabulary:
    The words and symbols in the language.
  - Syntax:
    The rules for combining words into statements.
  - Semantics:
    Define the rules for interpreting statements.

3.1 Language Elements

- Table 3-1 displays some Java vocabulary

TYPE OF ELEMENT	EXAMPLES
arithmetic operators	+ - * /
assignment operator	=
numeric literals	5.73 9
programmer defined variable names	fahrenheit celsius

3.1 Language Elements

- Programming Languages vs. Natural Languages:

Size:

- Programming languages have small vocabularies and simple syntax and semantics.
- Basic elements are not hard to learn.

Rigidity:

- In programming languages, the syntax used must be absolutely correct.

Literalness:

- Since computers follow instructions in a very literal manner, a programmer must be exhaustively thorough. (You get what you ask for, which may not be what you wanted.)

3.2 Basic Java Syntax and Semantics

Data Types:

- Primitive data types (numbers, characters, booleans)
  - Combined in expressions
  - Use operators (addition and multiplication)
- Objects
  - Are sent messages
  - Must be instantiated before use
- Strings
  - Are objects
  - Are sent messages
  - Do not need to be instantiated
  - Can be combined using the *concatenation operator*

3.2 Basic Java Syntax and Semantics

Syntax:

- Primitive Data Types
  - Combined in expressions using operators
- Objects
  - Sent messages
  - Must be instantiated before used (except Strings)
  - Strings can be combined using the concatenation operator

3.2 Basic Java Syntax and Semantics

Numeric Data Types:

- Six **numeric** data types are used in Java:
  - **int** (integer)
  - **double** (floating-point numbers or numbers with decimals)
  - short (not part of the AP subset)
  - long (not part of the AP subset)
  - byte (not part of the AP subset)
  - float (not part of the AP subset)

3.2 Basic Java Syntax and Semantics

- Table 3-2 shows some Java numeric data types:

TYPE	STORAGE REQUIREMENTS	RANGE
int	4 bytes	-2,147,483,648 to 2,147,483,647
double	8 bytes	-1.79769313486231570E+308 to 1.79769313486231570E+308

3.2 Basic Java Syntax and Semantics

Numeric Data Types:

- Programs that manipulate numeric data types often share a common format:
  - Input numeric data
  - Perform calculations
  - Output numeric results

3.2 Basic Java Syntax and Semantics

Literals:

- Literals are items in a program whose values do not change.

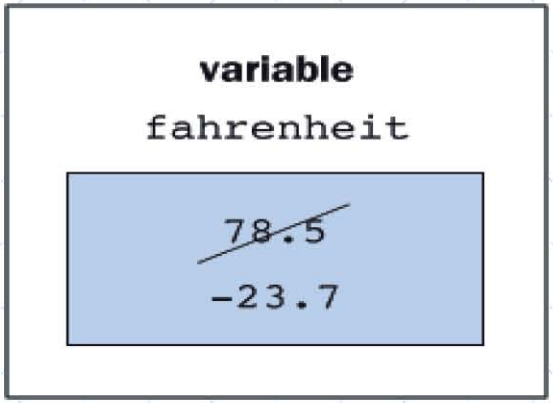
Table 3-3 lists some examples of numeric literals.

EXAMPLE	DATA TYPE
51	an integer
–31444843	a negative integer
3.14	a floating-point number (double)
5.301E5	a floating-point number equivalent to $5.301 * 10^5$, or 530,100
5.301E–5	a floating-point number equivalent to $5.301 * 10^{-5}$, or 0.00005301 (double)

3.2 Basic Java Syntax and Semantics

Variables and Their Declarations:

- A variable is an item whose value can change during the execution of a program.
- Changing the value of a variable is equivalent to replacing the value that was in the cell with another value.
- The type of data a variable contains cannot change.



3.2 Basic Java Syntax and Semantics

Declarations

- Variables
  - Before using a variable for the first time, the program must declare it's type.
  - Declare a variable in a ***variable declaration statement***

```
int age;
double celsius;
String name;
Scanner reader;
```

  - The type appears on the left and the variable's name on the right
    - celsius is a double

3.2 Basic Java Syntax and Semantics

- Several variables can be declared in a single declaration.
- Initial values can be assigned simultaneously to variable declarations:

```
int x,  y,  z = 7;
double p,  q = 1.41,  pi = 3.14,  t;
String name = "Bill Jones";
Scanner reader = new Scanner (System.in);
```

(class name object variable instantiate variable class name)

3.2 Basic Java Syntax and Semantics

- Objects
  - Declare the object variable **reader**, instantiate or create a **Scanner** object, and assign the object to the variable.
    - **new** <name of class>(zero or more parameters)
- Constants
  - The value of the variable cannot change
    - **final** double SALES_TAX_RATE =7.85;
  - "**final**" indicates a variable is declared as a ***constant***
  - Names of constants are written in UPPERCASE
  - Changing the value of a constant after it is initialized will be flagged by the compiler as an error.

3.2 Basic Java Syntax and Semantics

Assignment Statements

- An assignment statement has the following form:

  <variable> = <expression>;

- The value of the expression on the right is assigned to the variable on the left:

  fahrenheit = reader.nextDouble();

  name = "Bill Smith";

3.2 Basic Java Syntax and Semantics

Arithmetic Expressions

- An ***arithmetic expression*** consists of operands and operators combined in a manner familiar from Algebra. The usual rules apply:
  - Multiplication and division are evaluated before addition and subtraction.
  - Operators of equal precedence are evaluated from left to right.
  - Parentheses can be used to change the order of evaluation.

3.2 Basic Java Syntax and Semantics

- Multiplication must be indicated explicitly (a * b cannot be written as ab or (a)(b))
- Binary operators are placed between their operands (a * b)
- Unary operators are placed before their operands (-a)

TYPE	EXAMPLE
Literals	`32.0  5.0  9.0`
Variables	`fahrenheit  celsius`
Parenthesized expressions	`(fahrenheit - 32.0)`

3.2 Basic Java Syntax and Semantics

Common operators and their precedence:

OPERATOR	SYMBOL	PRECEDENCE (FROM HIGHEST TO LOWEST)	ASSOCIATION
Grouping	( )	1	Not applicable
Method selector	.	2	Left to right
Unary plus	+	3	Not applicable
Unary minus	-	3	Not applicable
Instantiation	new	3	Right to left
Cast	(double)	3	Right to left
	(int)		
Multiplication	*	4	Left to right

3.2 Basic Java Syntax and Semantics

OPERATOR	SYMBOL	PRECEDENCE (FROM HIGHEST TO LOWEST)	ASSOCIATION
Division	/	4	Left to right
Remainder or modulus	%	4	Left to right
Addition	+	5	Left to right
Subtraction	-	5	Left to right
Assignment	=	10	Right to left

3.2 Basic Java Syntax and Semantics

- Division:

  Several points concerning operators need explanation. First, the semantics of division are different for integer and floating-point operands. Thus:

  - 5.0/2.0 yields 2.5
  - 5/2 yields 2 (a quotient in which the fractional portion of the answer is simply dropped)

- Modulus:

  The operator % yields the remainder obtained when one number is divided by another. Thus:

  - 9 % 5 yields 4
  - 9.3 % 5.1 yields 4.2

3.2 Basic Java Syntax and Semantics

- Precedence:

  When evaluating an expression, Java applies operators of higher precedence before those of lower precedence unless overridden by parentheses.

  - 3+5*3 yields 18
  - -3+5*3 yields 12
  - +3+5*3 yields 18 (use of unary + is uncommon)
  - 3+5*-3 yields -12
  - 3+5*+3 yields 18 (use of unary + is uncommon)
  - (3+5)*3 yields 24
  - 3+5 % 3 yields 5
  - (3+5) % 3 yields 2

3.2 Basic Java Syntax and Semantics

Association:

The column labeled "Association" in Table 3-5 indicates the order in which to perform operations of equal precedence. Thus:

- 18-3-4 yields 11
- 18/3*4 yields 24
- 18 % 3*4 yields 0
- a=b=7; assigns 7 to b and b to a

More Examples

More examples of expressions and their values are shown in Table 3-6. In this table, we see the application of two fairly obvious rules governing the use of parentheses

- Parentheses must occur in matching pairs
- Parenthetical expressions may be nested but must not overlap.

3.2 Basic Java Syntax and Semantics

EXPRESSION	SAME AS	VALUE
3 + 4 - 5	7 - 5	2
3 + (4 - 5)	3 + (-1)	2
3 + 4 * 5	3 + 20	23
(3 + 4) * 5	7 * 5	35
8 / 2 + 6	4 + 6	10
8 / (2 + 6)	8 / 8	1
10 - 3 - 4 - 1	7 - 4 - 1	2
10 - (3 - 4 - 1)	10 - (-2)	12
(15 + 9) / (3 + 1)	24 / 4	6
15 + 9 / 3 + 1	15 + 3 + 1	19
(15 + 9) / ((3 + 1) * 2)	24 / (4 * 2) 24 / 8	3
(15 + 9) / (3 + 1) * 2	24 / 4 * 2 6 * 2	12

3.2 Basic Java Syntax and Semantics

- **The largest & smallest integers:**
  - Integer.MAX_VALUE
    - 2,147,483,647
  - Integer.MIN_VALUE
    - - 2,147,483,648
- **Arithmetic overflow error :**
  Assigning a value to a variable that is outside of the ranges of values that the data type can represent.

3.2 Basic Java Syntax and Semantics

Mixed-Mode Arithmetic

- Intermixing integers and floating-point numbers is called ***mixed-mode arithmetic.***
- When binary operations occur on operands of different numeric types, the less inclusive type (**int**) is temporarily and automatically converted to the more inclusive type (**double**) before the operation is performed.

3.2 Basic Java Syntax and Semantics

- Mixed-mode assignments are also allowed, provided the variable on the left is of a more inclusive type than the expression on the right. Otherwise, a syntax error occurs.
  - double d;
  - int i;
  - i = 45; --OK, because we assign an int to an int
  - d = i; --OK, because d is more inclusive than i. The value 45.0 is stored in d.
  - i = d; --Syntax error because i is less inclusive than d.
- Difficulties associated with mixed-mode arithmetic can be circumvented using a technique called "casting". This allows one data type to be explicitly converted to another type.

3.2 Basic Java Syntax and Semantics

- **Type casting:** Temporarily converting one data type to another
  - Can type cast a single variable or an entire expression
  - Place the desired data type within parentheses before the variable or expression that will be cast to another data type.
  - When casting an expression place parentheses around both the data type and the expression.
    - `int x = (int)(d + 1.6);`

3.2 Basic Java Syntax and Semantics

String Expressions and Methods

- Simple Concatenation
  - The concatenation operator uses the plus symbol (+)

```
String  firstName,                          //declare four string
        lastName,                           //variables
        fullName,
        lastThenFirst;

firstName = "Bill";                         //initialize firstName
lastName = "Smith";                         //initialize lastName

fullName = firstName +" " + lastName;   //yields "Bill Smith"
lastThenFirst = lastName +", "+ firstName;  //yields "Smith, Bill"
```

3.2 Basic Java Syntax and Semantics

- Concatenating Strings and Numbers
  - Strings also can be concatenated to numbers. (The number is automatically converted to a string before the concatenation operator is applied.)

```
String message;
int x = 20, y = 35;

message = "Bill sold " + x + " and Sylvia sold "
+ y + " subscriptions.";
// yields "Bill sold 20 and Sylvia sold 35 subscriptions."
```

3.2 Basic Java Syntax and Semantics

- Precedence of Concatenation
  - The concatenation operator has the same precedence as addition, which can lead to unexpected results:

"number " + 3 + 4 -> "number 3" + 4 -> "number 34"
"number " + (3 + 4) -> "number " + 7 -> "number 7"
"number " + 3 * 4 -> "number " + 12 -> "number 12"
3 + 4 + " number" -> 7 + " number" -> "7 number"

3.2 Basic Java Syntax and Semantics

- Escape Character
  - String literals are delimited by quotation marks ("..."), which presents a dilemma when quotation marks are supposed to appear inside a string.
  - Placing a special character before the quotation mark, indicating the quotation mark is to be taken literally and not as a delimiter, solves the problem.
  - This special character, also called the ***escape character***, is a backslash (\).

```
Message = "As the train left the station, " +
          "the conductor yelled, \"All aboard.\"";
```

3.2 Basic Java Syntax and Semantics

- Escape Character
  - The escape character also is used when including other special characters in string literals.
  - Special sequences involving the backslash character are called ***escape sequences***
    - Backslash t (\t) indicates a tab character
    - Backslash n (\n) indicates a newline character
  - When a string must contain a backslash, use two backlashes in sequence to escape the ***escape character.***
    - Path = "c:\\Java\\Ch3.doc";
      yields the string C:\Java\Ch3.doc

3.2 Basic Java Syntax and Semantics

- The length Method
  - Strings are objects and implement several methods.
  - A string returns its length in response to a length message:

```
String theString;
int theLength;

theString = "The cat sat on the mat.";
theLength = theString.length();    // yields 23
```

3.2 Basic Java Syntax and Semantics

Methods, Messages, and Signatures

- Classes implement methods, and objects are instances of classes.
- An object responds to a message only if its class implements a corresponding method.
- To correspond the method must have the same name as the message.
  - Messages are sometimes accompanied by parameters and sometimes not:

```
double x = reader.nextDouble();  // No parameter expected
System.out.println(50.5);        // One parameter expected
```

3.2 Basic Java Syntax and Semantics

- The parameters included when a message is sent must match exactly in number and type the parameters expected by the method.

```
double d = 24.6;
Math.sqrt (d);          // Perfect! A parameter of type double is expected
Math.sqrt (2.0 * d);    // Perfect! The expression yields a double.
Math.sqrt (4);          // Fine! Integers can stand in for doubles.
Math.sqrt ( );          // Error! A parameter is needed.
Math.sqrt (6.7, 3.4); // Error! One parameter only please.
Math.sqrt ("far");      // Error! A string parameter is NOT acceptable.
```

3.2 Basic Java Syntax and Semantics

- Some methods return a value and others do not.
- To use a method successfully we must know:
  - What type of value it returns
  - Its name
  - The number and type of the parameters it expects
- This information is called the method's ***signature.***

3.2 Basic Java Syntax and Semantics

- User-Defined Symbols
  - Must begin with a letter of the alphabet
    - A ... Z
    - a ... z
    - _ and $ (I recommend that you don't begin a user defined symbol with these.)
  - Can include other letters and / or digits.
  - Cannot include a space.
    - Use the underscore character instead of a space.
    - i.e. symbol_Name

3.2 Basic Java Syntax and Semantics

- Keywords
  - ***Keywords*** or ***reserved words*** cannot be employed as user-defined symbols because they have special meaning in Java.
  - Keywords are also case sensitive. "import" is a reserved word but "Import" and "IMPORT" are not.

3.2 Basic Java Syntax and Semantics

Table 3-7 displays a list of Java's reserved words

abstract	double	int	static
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	null	throw
char	for	package	throws
class	goto	private	transient
const	if	protected	try
continue	implements	public	void
default	import	return	volatile
do	instanceof	short	while

3.2 Basic Java Syntax and Semantics

- Programming Protocols:
  - Well-chosen variables names greatly increase a program's readability and maintainability
  - It is considered good programming practice to use meaningful names such as:

    radius rather than r

    taxableIncome rather than ti
  - Examples of valid and invalid variable names:

    Valid Names: surfaceArea3 _$_$$$

    Invalid Names: 3rdPayment pay.rate abstract

3.2 Basic Java Syntax and Semantics

- Programming Protocols:
  - When forming a compound variable name, programmers usually capitalize the first letter of each word except the first.

    (For example: taxableIncome)
  - All the words in a program's name typically begin with a capital letter (ComputeEmployeePayroll).
  - Constant names usually are all uppercase (CONSTANT_NAME).

3.2 Basic Java Syntax and Semantics

Packages and the import statement

- Java often utilizes code written by many other programmers.
- A ***package*** makes it easy for programmers to share code.
- A programmer can collect the classes together in a package, and then import classes from the package.
- The Java programming environment typically includes a large number of standard packages.
- When using a package, a programmer imports the desired class or classes.

3.2 Basic Java Syntax and Semantics

- The general form of an import statement is:

  import x.y.z;

  where

  x is the overall name of the package.

  y is the name of a subsection within the package.

  z is the particular class in the subsection.
- It is possible to import all the classes within a subsection at once.
- The statement to import all the classes within a subsection looks like this:

  import x.y.*;
- A star (*) is used to make available all of the classes in a package.

3.3 Terminal I/O for Different Data types

- Objects support terminal input and output.
- An instance of the class Scanner supports input.
- The object System.out supports output.
- Object System.out is an instance of the class PrintStream.
  - The class PrintStream, is available to Java programmers without specifying a name in an import statement.
  - However, the Scanner class requires importing the package: import java.util.Scanner;

3.3 Terminal I/O for Different Data types

Table 3-8 summarizes the methods in class Scanner.

METHOD	DESCRIPTION
`double nextDouble()`	Returns the first `double` in the input line. Leading and trailing spaces are ignored.
`int nextInt()`	Returns the first integer in the input line. Leading and trailing spaces are ignored.
`String nextLine()`	Returns the input line, including leading and trailing spaces. *Warning*: A leading newline is returned as an empty string.

3.3 Terminal I/O for Different Data types

The following program illustrates the major features of terminal I/O: TestTerminalIO.java TestTerminalIO.txt

```
import java.util.Scanner;

public class TestTerminalIO {
   public static void main (String [] args) {
      Scanner reader = new Scanner(System.in);
      String name;
      int age;
      double weight;

      System.out.print ("Enter your name (a string): ");
      name = reader.nextLine();

      System.out.print ("Enter your age (an integer): ");
      age = reader.nextInt();
```

3.3 Terminal I/O for Different Data types

```
    System.out.print ("Enter you weight (a double): ");
    weight = reader.nextDouble();

    System.out.println ("Greetings " + name +
                        ". You are " + age +
                        " years old and you weigh " + weight + "
pounds.");

  }
}
```

String Errors using nextLine()

- Look at the following program:
  - TestTerminalIOWithError.java
  - TestTerminalIOWithError.txt
- If you attempt to read a string from the input stream after an integer or double has been entered the string will be empty.
  - The methods nextInt() and nextDouble() ignore and do NOT consume the newline character that the user entered following the number.
  - The newline character was waiting in the input stream to be consumed by the nextLine() method, which was expecting more data.

Correcting nextLine() Errors

- To correct this error, add another reader.nextLine(); statement to consume the newline character **before** reading the string from the input stream.
- Here is the corrected code:
  - TestTerminalIOWithErrorFixed.java
  - TestTerminalIOWithErrorFixed.txt

3.4 Comments

- ***Comments*** are explanatory sentences inserted in a program in such a matter that the compiler ignores them.
- There are two styles for indicating comments:
  - End of line comments:

    These include all of the text following a double slash (//) on any given line; in other words, this style is best for just one line of comments
  - Multiline comments:

    These include all of the text between an opening /* and a closing */

3.4 Comments

- The following code segment illustrates the use of both kinds of comments.

```
/* This code segment illustrates the
use of assignment statements and comments */

a = 3;        // assign 3 to variable a
b = 4;        // assign 4 to variable b
c = a + b;    // add the number in variable a
              //  to the number in variable b
              //  and assign the result, 7 , to variable c
```

3.4 Comments

- The main purpose of comments is to make a program more readable and thus easier to maintain.
- One should:
  - Begin a program with a statement of its purpose and other information that would help orient a programmer called on to modify the program at some future date.
  - Accompany a variable declaration with a comment that explains the variable's purpose.
  - Precede major segments of code with brief comments that explain their purpose.
  - Include comments to explain the workings of complex or tricky sections of code.

3.4 Comments

- Too many comments are as harmful as too few, because over time, the burden of maintaining the comments becomes excessive.
  - Don't use comments that state the obvious.
- The best written programs are self-documenting; that is, the reader can understand the code from the symbols used and from the structure and overall organization of the program.

Case Study 1

Income Tax Calculator.java

Income Tax Calculator.txt

3.5 Programming Errors

The Three Types of Errors

- ***Syntax errors***
  - Occur when a syntax rule is violated (no matter how minor)
  - Are detected at compile time.
  - When the Java compiler finds a syntax error, it prints an error message.
  - Error messages are often quite cryptic.

3.5 Programming Errors

- ***Run-time errors***
  - Occur when the computer is asked to do something that it considers illegal, (such as dividing by zero)
  - x/y is syntactically correct
  - When the expression is evaluated during execution of the program, the meaning of the expression depends on the values contained in the variables.
    - (If the variable y has the value 0, then the expression cannot be evaluated)
  - The Java run-time environment will print a message telling us the nature of the error and where it was encountered.
  - The error message might be hard to understand.

3.5 Programming Errors

- ***Logic errors*** (*design errors* or *bugs*)
  - Occur when we fail to express ourselves accurately.
    - The instruction is phrased properly, and thus the syntax is correct.
    - The instruction is meaningful, and thus the semantics are valid.
    - But the instruction does not do what we intended, and thus is logically incorrect.
  - Programming environments do not detect logic errors automatically.

Errors

- DivideByIntegerZero.java
  - DivideByIntegerZero.txt
- DivideByFloatingPointZero.java
  - DivideByFloatingPointZero.txt
- PuzzlingRunTimeError.java
  - PuzzlingRunTimeError.txt

3.6 Debugging

- A bug is not always easy to locate.
- Often bugs are not located where one might expect them.
- Adding extra lines to the program can help to locate a bug.
- Determining if any of the variables deviate from their expected values will highlight the existence of a bug.
- A variables value is printed in the terminal window as follows:

```
System.out.println ("<some message>" +
        <variable name>);
```

3.6 Debugging

- The following program claims that 212 degrees Fahrenheit converts to 41.1 degrees Celsius instead of the expected 100.
- Try checking the value of fahrenheit just before celsius is calculated. The needed code looks like this:

```
System.out.println ("fahrenheit = " + fahrenheit);     // This is
        the debugging code
celsius = (fahrenheit  - 32.0) * 5.0 / 9.0;
```

3.6 Debugging

- When the program runs again with the debugging code included, we get the following output:

```
Enter degrees Fahrenheit: 212
Fahrenheit = 106.0
The equivalent in celsius is 41.111111111111114
```

- 212 is entered but for some reason, the program says the value of fahrenheit is 106.

3.6 Debugging

- Examine the surrounding code to try to spot the error.

```
. . .
System.out.print ("Enter degrees Fahrenheit: ");
fahrenheit = reader.nextDouble( ) / 2.0;
System.out.println ("fahrenheit = " + fahrenheit);
celsius = (fahrenheit – 32.0) * 5.0 / 9.0;
. . .
```

- The error is that the value entered by the user is divided by 2 just before it is assigned to the variable fahrenheit.

Case Study 2

CountAngels.java

CountAngels.txt

Summary

- Use the `int` data type for whole numbers and `double` for floating-point numbers.
- Variable and method names consist of a letter followed by additional letters or digits.
- Keywords cannot be used as names.
- Final variables behave as constants; their values cannot change after they are declared.

Summary (cont.) 1

- Arithmetic expressions are evaluated according to precedence.
- Some expressions yield different results for integer and floating-point operands.
- Strings may be concatenated.
- The compiler catches syntax errors.
- The JVM catches run-time errors.

Summary (cont.) 2

- Logic errors, if caught, are detected by the programmer or user at run-time.
- Can find and remove logic errors by inserting debugging output statements to view the values of variables.