UCLA PIC 20A
Java Programming

•**Instructor:** **Ivo Dinov,**
Asst. Prof. In Statistics, Neurology and Program in Computing

•**Teaching Assistant:** Yon Seo Kim, **PIC**

University of California, Los Angeles, Summer 2002
http://www.stat.ucla.edu/~dinov/

Chapter 3 – Java Basics

- Variables
- Operators
- Expressions, Statements, and Blocks
- Control Flow Statements

Variables

- **An object stores its state in variables.**
- A ***variable*** is an item of data named by an identifier.
- You must explicitly provide a name and a type for each variable you want to use in your program. The variable's name must be a legal ***identifier***—an unlimited series of Unicode 1 characters that begins with a letter.
- You use the variable name to refer to the data that the variable contains. The variable's type determines what values it can hold and what operations can be performed on it. To give a variable a type and a name, you write a variable *declaration*, which generally looks like this:

type name

Variables

- In addition to the name and the type that you explicitly give a variable, a variable has ***scope***.
- Ex. MaxVariablesDemo, (JavaTutorialExamples\java\nutsandbolts\example-1dot1)

```
public class MaxVariablesDemo {
public static void main(String args []){
    byte largestByte =Byte.MAX_VALUE;
    short largestShort =Short.MAX_VALUE;
    int largestInteger =Integer.MAX_VALUE;
    long largestLong =Long.MAX_VALUE;
        //real numbers
    float largestFloat =Float.MAX_VALUE;
    double largestDouble =Double.MAX_VALUE;
```


Variables

- In addition to the name and the type that you explicitly give a variable, a variable has ***scope***.
- Ex. MaxVariablesDemo, (JavaTutorialExamples\java\nutsandbolts\example-1dot1)

The output from this program is:

The largest byte value is	127
The largest short value is	32767
The largest integer value is	2147483647
The largest long value is	9223372036854775807
The largest float value is	3.40282e+38
The largest double value is	1.79769e+308
The character S is	upper case
The value of a Boolean is	true

Data types

- A variable's data type determines the values that the variable can contain and the operations that can be performed on it. For example, the declaration **int largestInteger** declares that largest Integer has an integer data type.
- Java has two categories of data types: **primitive** *(*single value of the appropriate size and format*)* and **reference** (Arrays, classes, and interfaces. The value of a reference type variable is a reference (pointer) to an address of the value, or set of values, represented by the variable.)
- Java does not support the explicit use of addresses like C/C++. You use the variable's name instead.

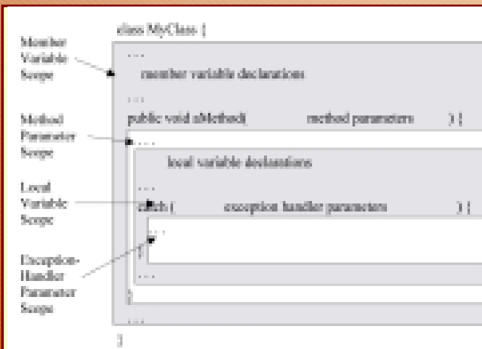
Variable Names

- A program refers to a variable's value by the variable's name. For example, when it displays the value of the **largestByte** variable, the **MaxVariablesDemo** program uses the name **largestByte**. A name, such as **largestByte**, that's composed of a single identifier, is called a *simple name*. Simple names are in contrast to *qualified* names, which a class uses to refer to a member variable that's in another object or class.
- For a **simple name**:
  - It must be a legal identifier (starting with a letter).
  - It must not be a keyword
  - It must be unique within its scope.

Variable Names

- **By Convention: Variable names** begin with a **lowercase letter**, and **class names** begin with an **uppercase letter**. If a variable name consists of more than one word, the words are joined together, and each word after the first begins with an uppercase letter, e.g.,: **isThisObjectVisible** . The underscore character (_) is acceptable anywhere in a name, but by convention is used only to separate words in constants (because constants are all caps by convention and thus cannot be case-delimited).

Variable Scope



Variable Initialization

- Local variables and member variables can be initialized with an assignment statement when they're declared.
- The data type of the variable must match the data type of the value assigned to it.
- The **MaxVariablesDemo** program provides initial values for all its local variables when they are declared.
  - // integers
  - byte largestByte =**Byte.MAX_VALUE** ;
  - short largestShort = **23000;** // **Short.MAX_VALUE** ;

Final Variable

- You can declare a variable in any scope to be final. The value of a final variable cannot change after it has been initialized. Such variables are similar to constants in other languages.
- To declare a final variable, use the final keyword in the variable declaration before the type:

```
final int aFinalVar =0;
```

```
final int   blankFinal;
...
blankFinal = 0;
```

- The previous statement declares a final variable and initializes it, all at once. Subsequent attempts to assign a value to **aFinalVar** result in a compiler error. Again, once a final local variable has been initialized, it cannot be set again.

Arithmetic Operations

- Java supports various arithmetic operators for all floating-point and integer numbers. These operators are:

Operator	Use	Description
+	op1+op2	Adds op1 and op2 ;
-	op1-op2	Subtracts op2 from op1
*	op1*op2	Multiplies op1 by op2
/	op1/op2	Divides op1 by op2
%	op1 %op2	rem of dividing op1 by op2

- Ex: **java\nutsandbolts\example\ArithmeticDemo.java**

Unitary Arithmetic Operators

- Two shortcut arithmetic operators are ++, which increments its operand by 1, and --, which decrements its operand by 1. Either ++ or -- can appear before *(prefix)* or after *(postfix)* its operand. The prefix version, ++op /--op , evaluates to the value of the operand *after* the increment. The postfix version, op++/--, evaluates to the value of the operand *before* the increment/decrement operation.

Operator	Use	Description
+	+op	Promotes op to int if it's a byte , short , or char
-	-op	Arithmetically negates op
++	++op	(or op++) op = op+1
--	--op	(or op--) op = op -1

Relational & Conditional Operators

Operator	Use	Description
>	op1 >op2	Returns true if op1 is greater than op2
>=	op1 >=op2	Returns true if op1 >= op2
<	op1 <op2	Returns true if op1 is less than op2
<=	op1 <=op2	Returns true if op1 <= op2
==	op1 ==op2	Returns true if op1 and op2 are equal
!=	op1 !=op2	True if op1 and op2 are not equal

- Ex. RelationalDemo.java

Relational & Conditional Operators

Operator — Use — Description

- && — op1 && op2 — Returns true if op1 and op2 are both true ;
- || — op1 ||op2 — Returns true if either op1 or op2 is true ;
- !! — op — Returns true if op is false
- & — op1 & op2 — Returns true if op1 and op2 are both boolean and true. If both operands are numbers, performs bitwise AND operation
- | — op1 |op2 — true if both op1 and op2 are boolean, and either is true. If both operands are numbers, performs bitwise inclusive OR
- ^ — op1 ^ op2 — true if op1 and op2 are different, that is, if one or the other of the operands, but not both, is true

- Ex. RelationalDemo.java

Shift & Bitwise Operators

- A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left.

Operator — Use — Description

<< — op1 <<op2 — Shift bits of op1 left by distance op2 ; fills with zero bits on the right

>> — op1 >>op2 — Shift bits of op1 right by distance op2 ; fills with highest (sign) bit on the left

>>> — op1 >>>op2 — Shift bits of op1 right by distance op2 ; fills with zero bits on the left-hand side

- Ex. RelationalDemo.java

Shift & Bitwise Operators

- Example: 13 >>1;

The binary representation of the number 13 is 1101. The result of the shift operation is 1101 shifted to the right by one position—110, or 6 in decimal. The left-hand bits are filled with 0's as needed.

Shift & Bitwise Operators

- A shift operator performs bit manipulation on data by shifting the bits of its first operand right or left.

Operator — Use — Description

- & — op1& op2 — Bitwise AND , if both operands are numbers. Conditional AND, if operands are boolean
- | — op1 |op2 — Bitwise inclusive OR , if operands are numbers. Conditional OR, if operands are boolean
- ^ — op1 ^ op2 — Bitwise exclusive OR (XOR)
- ~ — ~op2 — Bitwise complement

- Ex. RelationalDemo.java

Shift & Bitwise Operators

- See the tables for { & | ^ ~ } in the textbook.
  - ~ 1011_2 (11_{10}) → is 0100_2 (4_{10})
  - 1101_2 (13_{10}) & 1100_2 (12_{10}) → is 1100_2 (12_{10})
  - 1101_2 (13_{10}) ^ 1100_2 (12_{10}) → is 1_2 (1_{10})
  - 1001_2 (9_{10}) | 1100_2 (12_{10}) → is 1101_2 (13_{10})

Expressions

- Expressions perform the work of a program. Among other things, expressions are used to compute and to assign values to variables and to help control the execution flow of a program.
- The job of an expression is twofold: to perform the computation indicated by the elements of the expression and to return a value that is the result of the computation.
- **Definition:** An ***expression*** is a series of variables, operators, and method calls (constructed according to the syntax of the language) that evaluates to a single value. As discussed in the previous section, operators return a value, so the use of an operator is an expression. Ex: **MaxVariablesDemo**

Operator Precedence

- High Precedence:

Postfix operators	[] ..(*params*)*expr* ++*expr* --
Unary operators	++*expr* --*expr* +*expr* -*expr* ~!
Creation or cast new	(*type*)*expr*
Multiplicative	*/%
Additive	+-
Shift	>>>>
Relational	<><=>= instanceof
Equality	==!=
Bitwise AND	&
Bitwise exclusive OR	^
Bitwise inclusive OR		
Conditional AND	&&
Conditional OR	||
Shortcut if -	?:
Assignment	=+=-=*=/=%=&=^=|=<<=>>=>>>=

- Low Precedence

Statements

- Statements are roughly equivalent to sentences in natural languages. **A statement forms a complete unit of execution**.
- Expression Statements:
  - aValue =8933.234; // assignment statement
  - aValue++; // increment statement
  - System.out.println(aValue); // method call statement
  - Integer intObject =new Integer(4); //object creation statement
- Declaration statement: **double aValue =8933.234;**
- Control flow statement: regulates the order in which statements get executed. The **for** loop and the **if** statement.

Blocks (of statements)

- A ***block*** is a group of zero or more statements between balanced braces and can be used anywhere a single statement is allowed.
- Ex. **MaxVariablesDemo** program:

2 Blocks: if & else

```
if (Character.isUpperCase(aChar)){
    System.out.println(aChar +"is upper case.");
}  else  {
    System.out.println(aChar +"is lower case.");
}
```


Control-flow statements

- ***Control flow statements*** allow conditional execution
- For example, in the following code snippet, the if statement conditionally executes the System.out.println statement within the braces, based on the return value of Character.isUpperCase(aChar):

```
char c;
...
if (Character.isUpperCase(aChar)){
    System.out.println(aChar +"is upper case.");
}
```

Control-flow statements

- Java provides several control flow statements
- **Statement Type** — **Keywords**

Statement Type	Keywords
Looping	**while , do-while , for**
Decision making	**if-else , switch-case**
Exception handling	**try-catch-finally , throw**
Branching	**break, continue, label:, return**

Control-flow statements – while / do loops

while (*expression*){

statement(s)

}

```
while (c !='g')  {
        copyToMe.append(c);
        c
=copyFromMe.charAt(++i);
}
```

do {

statement(s)

} while (*expression*);

```
do  {
     copyToMe.append(c);
     c =copyFromMe.charAt(++i);
}  while (c !='g');
```


Control-flow – *if-else* conditioning

```
if (expression1){
      statement(s)
}
else if (expression2){
      statement(s)
}
...
else {
      statement(s)
}
```

IfElseDemo.java

```
if (testscore >=90){
        grade ='A';
} else if (testscore >=80) {
        grade ='B';
}  else if (testscore >=70){
        grade ='C';
}  else if (testscore >=60){
        grade ='D';
}  else {
        grade ='F';
}
System.out.println("GRD="+grade);
```


Control-flow – *switch* conditioning

SwitchDemo.java

```
switch (month){
        case 1:System.out.println("January");break;
        case 2:System.out.println("February");break;
        case 3:System.out.println("March");break;
        case 4:System.out.println("April");break;
        case 5:System.out.println("May");break;
        case 6:System.out.println("June");break;
        case 7:System.out.println("July");break;
        case 8:System.out.println("August");break;
        case 9:System.out.println("September");break;
        case 10:System.out.println("October");break;
        case 11:System.out.println("November");break;
        case 12:System.out.println("December");break;
}
```


Exception Handling

- **Java provides a mechanism known as *exceptions* to help programs report and handle errors. When a run-time error occurs, the program throws an exception. This means that the normal flow of the program is interrupted and that the runtime environment attempts to find an *exception handler*—a block of code that can deal with a particular type of error. The exception handler can attempt to recover from the error or, if it determines that the error is unrecoverable, provide a gentle exit from the program.**
- **Three statements play a part in handling exceptions.**
  - **Try**
  - **Catch**
  - **Finally**

Exception Handling

- **try** statement identifies a block of statements within which an exception might be thrown.
- **catch** statement must be associated with a try statement and identifies a block of Statements that can handle a particular type of exception. The statements are executed if an exception of a particular type occurs within the try block.
- **finally** statement must be associated with a try statement and identifies a block of statements that are executed regardless of whether an error occurs within the try block.

Exception Handling

```
try {
        statement(s)
  } catch (exceptiontype name ) {
        statement(s)
  } finally {
        statement(s)
  }
```


Branching Statements

- The **break** statement
- The **continue** statement
- The **return** statement

The break statement and the continue statement can be used with or without a label. A *label* is an identifier placed before a statement. The label is followed by a colon (......):

statementName: someJavaStatement;

Ex:
```
for ( int i = 0 ;  i < arrayOfInts.length;  i++ )    {
    if (arrayOfInts [i ]  == searchfor)  {
        foundIt =true;
        break;
    }
}
```


Branching Statements with a Label

```
int j =0, i =0;
boolean foundIt =false;
search:
for (;  i <arrayOfInts.length; i++)  {
    for (j =0;  j <arrayOfInts [i ].length;  j++)  {
        if (arrayOfInts [i ][j ] ==searchfor){
            foundIt =true;
            break search;
        }
    }
}
….
```


Continue Statements

- You use the **continue statement to skip the current iteration of a for, while, or do loop**.
- The unlabeled form skips to the end of the innermost loop's body and evaluates the boolean expression that controls the loop, basically skipping the remainder of this iteration of the loop.
- ```
  for (int i =0;i <max;i++) {  //interested only in p's
      if (searchMe.charAt(i)  !=  'p')
          continue;
          //process p's
      numPs++;
      searchMe.setCharAt(i,'P');
  }
  ```


Return Statements

- Use return to exit from the current method. The flow of control returns to the statement that follows the original method call. The return statement has two forms:

  return ++count; // return (++count);

  The data type of the value returned by return must match the type of the method's declared return value.
- When a method is declared void , use the form of return that doesn't return a value:

  return;
  ```