

Chapter 2:

Primitive Data and

Definite Loops

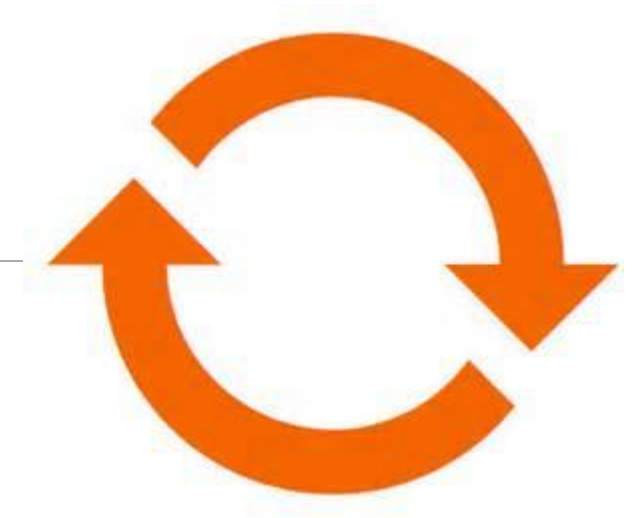
WEEK 2:

BASIC DATA CONCEPTS

VARIABLES

THE FOR LOOP

MANAGING COMPLEXITY: SCOPE & PSEUDOCODE



Basic Data Types

- Programs manipulate **INFORMATION**, which can come in many **TYPES**
- Java is a **TYPE-SAFE LANGUAGE**: You have to specify to Java what type of data you intend to manipulate
- Everything you manipulate in a Java program will be of a certain type
- **Data Type:**
 - A name for a **CATEGORY** of RELATED data values (e.g. type **int**, which is used to represent *integer* values)
- There are TWO different data types in Java:
 - **Primitive Data**
 - **Objects**

Primitive Types

- There are 8 primitive types in Java

FUNDAMENTAL & COMMONLY USED	EXIST FOR SPECIAL CIRCUMSTANCES
int	byte
double	float
char	long
boolean	short

- Type names are Java **KEYWORDS**
 - They are used in programs to let the computer know what data types you intend to use

- The two common **NUMERIC** data types, **int** and **double**, are represented in different ways in the computer's memory
- **Integers** are stored **EXACTLY**
- Numbers that include **FRACTIONAL** components (i.e. **doubles**) are stored as **APPROXIMATIONS**
 - They have a LIMITED number of digits for ACCURACY
- When working with **double** type data, you must watch out for “**round-off errors**”
- If it is **possible** to use the **int** type instead of the **double** type, it is **better** to do so

Expressions

- **Expression:** A simple **value** or a **set of operations** that produces a value
- **The Simplest Expressions:** Simple values such as 42 or 17.9
 - Referred to as LITERAL values or “**LITERALS**”
- **More Complex Expressions:** Created by **combining** simple values
 - Example: $(2 * 6) + (7 * 4) - 3$
- **Evaluation:** The process of obtaining the value of an expression
- You can use expressions in your programs and have the computer **evaluate** them:

Here, a **numerical expression** is entered as input to the **println()** function

```
// NOTE USE OF print() instead of println()  
// to suppress the newline character.  
System.out.print("The sum of 3 and 4 is ");  
System.out.println(3 + 4);
```

Problems @ Javadoc Declaration Console X

<terminated> SampleCode [Java Application] C:\Program Files\Java\jre7\bin
The sum of 3 and 4 is 7

Operators & Operands

- **Operators:** Special symbols used to indicate an operation to be performed on one or more values
- **Arithmetic Operators**

OPERATOR	MEANING	EXAMPLE	RESULT
+	Addition	3 + 8	11
-	Subtraction	58 – 28	30
*	Multiplication	7 * 6	42
/	Division	11.8 / 2.0	5.9
%	Remainder (“mod”)	27 % 5	2

- **Operands:** The values used in an expression
 - In the expression **3 + 29** , the values **3** and **29** are **operands** (and are separated by the **addition operator**)

Literals

- The simplest expressions refer to values DIRECTLY, using what are known as “**LITERALS**”
- **Numeric Information**
 - Integer literals (of type **int**)
 - A sequence of digits, with or without a leading sign
 - **Examples:** 3, 5, -5, 0, +72
 - Floating-point literals (of type **double**)
 - Any number that includes a decimal point is considered a **double**
 - You can express a large **double** in scientific notation, using e and a digit representing an exponent:
2.3e4 = 23000.0
 - **Examples:** 2.3e4, 298.4, 207. , -.98, +0.5
 - **Note:** A decimal point may be used with or without a leading or a following 0

Literals, cont.

- **Textual Information**

- Textual information can be stored in **string** literals – represented by enclosing the information in QUOTES
- **string** literals are composed of sequences of characters
- **string** literals may be composed of any legal characters – including digits
- **Examples:** “MyName”, “x”, “My name is x”, “5000”

- **Logical Information**

- The primitive type **boolean** stores logical information
- The **boolean** type has **TWO** possible values: **TRUE** or **FALSE**
- Often **boolean** primitives are used in programs to determine whether or not some condition has been met

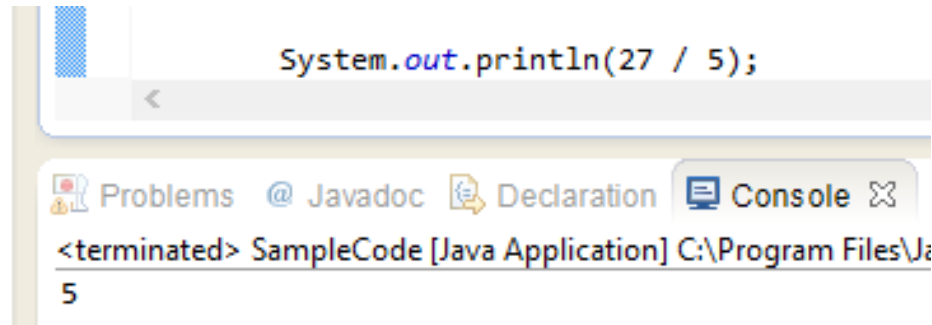


Division in Java

- **Integer Division**

- If both operands of a division operation are **integers**, Java will perform what is known as “**integer division**”
- This ensures that the result of the operation is **STILL an INTEGER**
- However, to guarantee this outcome, Java will **DISCARD** any fractional part on the result

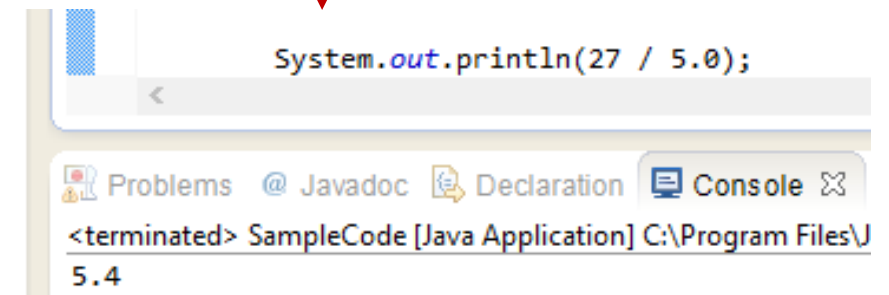
- **Example:**



A screenshot of an IDE window showing a Java code editor with the line `System.out.println(27 / 5);`. Below the editor, the 'Console' tab is active, displaying the output `<terminated> SampleCode [Java Application] C:\Program Files\J... 5`. A red arrow points from a note box to the denominator '5' in the code.

NOTE: See how the result changes when **5.0** is used in the denominator instead of **5**

- If you do **NOT** want to perform **integer division**, at least **ONE** of the operands must be of type **double**:



A screenshot of an IDE window showing a Java code editor with the line `System.out.println(27 / 5.0);`. Below the editor, the 'Console' tab is active, displaying the output `<terminated> SampleCode [Java Application] C:\Program Files\J... 5.4`. A red arrow points from the note box to the denominator '5.0' in the code.

The “mod” Operator (modulo)

- **mod operator**: used to capture the **remainder** that is left over after an integer division operation

- **Example: 19 % 3**

- First, set up a division problem:

$$\begin{array}{r} 6 \\ 3 \overline{) 19} \\ \underline{18} \\ 1 \end{array}$$

← Result of integer division

← Remainder after integer division &
Result of mod operation

- The amount “**left over**” after integer division (in this case **1**) is known as the **remainder**
- This **remainder** is the value **returned** by the **mod operator**: **19 % 3 = 1**

Division in Java: Special Cases

1. numerator < denominator

2. numerator of 0

3. denominator of 0

- We'll look at some Java code to see what happens
- Using a denominator of 0 results in an **ERROR** (both **10 / 0** and **10 % 0** give the same error):

```
The denominator is 0: 10/0
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at SampleCode.main(SampleCode.java:18)
```

Java statements:

```
System.out.println("The numerator is smaller than the denominator: 7 / 10 ");  
System.out.println(7 / 10);  
System.out.println("Now using the mod operator: 7 % 10 ");  
System.out.println(7 % 10);  
  
System.out.println("The numerator is 0: 0 / 10 ");  
System.out.println(0 / 10);  
System.out.println("Now using the mod operator: 0 % 10 ");  
System.out.println(0 % 10);
```

Output:

```
<terminated> SampleCode [Java Application] C:\Program Files\Java\jre7\bin\java.exe  
The numerator is smaller than the denominator: 7 / 10  
0  
Now using the mod operator: 7 % 10  
7  
The numerator is 0: 0 / 10  
0  
Now using the mod operator: 0 % 10  
0
```

Precedence

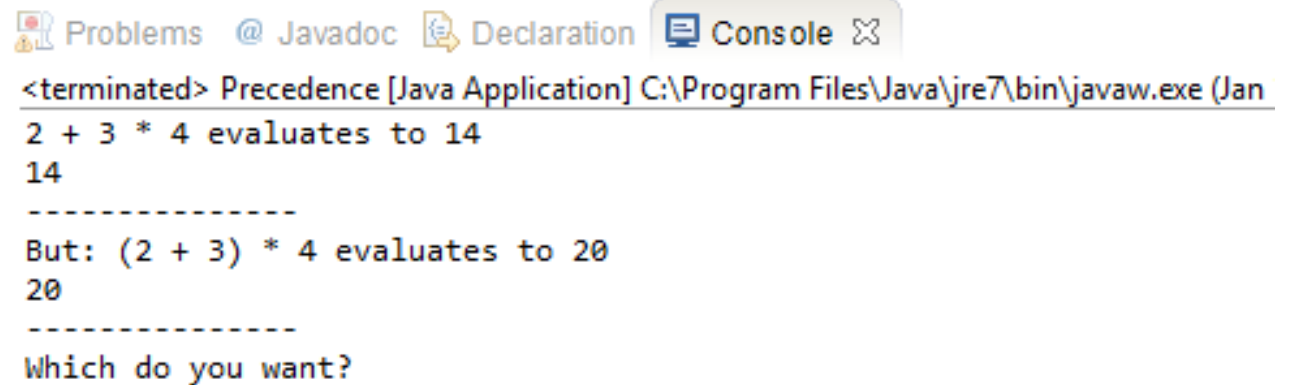
- There are rules that determine the order of evaluation of different parts of an expression
- In arithmetic expressions, “multiplicative” operations (using $*$, $/$, or $\%$) are evaluated before “additive” operations (using $+$, $-$)
- The multiplicative operators thus have “**higher precedence**” than the additive operators
- Unary (“sign”) operators have even higher precedence

Precedence	Description	Operators
Highest	Unary operators (positive/negative)	$+$, $-$ (for SIGN)
↓	Multiplicative operators	$*$, $/$, $\%$
Lowest	Additive operators	$+$, $-$

Precedence: Example

- You can control the order of operations in an expression by using PARENTHESES
- Group together the operands and operators you wish to combine
- Make sure your expressions are calculating the values you intended
- Even when parentheses are not necessary for the calculation to occur as intended, they can add **CLARITY** to your code for your **HUMAN** users

```
System.out.println("2 + 3 * 4 evaluates to 14");
System.out.println(2 + 3 * 4);
System.out.println("-----");
System.out.println("But: (2 + 3) * 4 evaluates to 20");
System.out.println((2 + 3) * 4);
System.out.println("-----");
System.out.println("Which do you want?");
```



The screenshot shows a Java IDE with a console window open. The console displays the output of the Java application, which is a demonstration of operator precedence. The output shows the result of the expression `2 + 3 * 4` as 14, followed by a separator line, then the result of the expression `(2 + 3) * 4` as 20, followed by another separator line, and finally the prompt "Which do you want?".

```
<terminated> Precedence [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan
2 + 3 * 4 evaluates to 14
14
-----
But: (2 + 3) * 4 evaluates to 20
20
-----
Which do you want?
```

Mixing Types and Casting

- Two Java types that are often “**mixed**” are the two common numeric types: **int** and **double**
- **Example:** Evaluating an expression such as **3.0 + 14**
- With such mixed type expressions, Java will do the **int** → **double** conversion automatically
 - The **result** returned by such a mixed type expression will be a **double**
- This is especially helpful when you have to divide numbers
 - If **either** of the operands is of type **double**, Java will do **real-valued division**, instead of integer division
 - **Example:** **7/2** → **3**, but all of the following evaluate to **3.5**: **7.0/2** **7/2.0** **7./2** **7/.2** **7.0/2.0** **7./2.**
- What if you want to convert **double** → **int** ?
- You can ask Java to do the conversion by using a “**cast**”, which will convert the **type** of the value
 - In a **double** → **int** conversion, this means **DISCARDING** – **NOT ROUNDING** -- any fractional component):

(int)5.2 → **5** and **(int) 7.9** → **7**

Variables

- **Variable:** A **named memory location** that **stores a value**
- When you create a variable in Java, you need to give the variable a **name** and you need to tell Java **the type of data** you intend to store
- Java will reserve a location in memory for the type of data it now expects to get
- Once you actually store some data, you have the option of **CHANGING** its value at a later point
- In the code example shown, **three new variables** are created
- The values of **x** and **y** are **changed** from their initial values later in the program
- The initial value stored in **z** depends on the values stored in **x** and **y** at the time **z** is **created**
- Once this initial value for **z** is stored, changes in **x** and/or **y** do not affect it

```
// Create 3 integer variables
int x = 10;
int y = 20;
int z = x + y;

// Change the values stored for x and y
x = 11;
y = 15;

System.out.print(" The value of x is: ");
System.out.println(x);
System.out.print(" The value of y is: ");
System.out.println(y);
System.out.print(" The value of z is: ");
System.out.println(z);
```

Problems @ Javadoc Declaration Console

<terminated> declareError [Java Application] C:\Program Files\Java\jre7\bin\java.exe

The value of x is: 11
The value of y is: 15
The value of z is: 30

Declaring Variables

- **Declaration:** A request to set aside a **new variable** with a given **name** and **type**
- Simple declarations are of the form: **<type> <name>;**
- **Examples:**

```
int counter;  
double weight;
```
- These simple variable declarations cause Java to set aside memory locations to store values of the specified types
- The variables are currently **UNINITIALIZED** – this means they have not yet been ASSIGNED values
- Values are associated with variables via **ASSIGNMENT STATEMENTS**
- The general syntax of an **assignment statement** is: **<variable> = <expression>;**
- **Examples:**

```
counter = 0;  
weight = 10 * 17.3;
```

More on Declaring Variables

- Each variable is declared just **ONCE** – if you try to declare a variable multiple times, you will get an **ERROR** message

Notice how the Java compiler tries to warn you of the error **AS** you type your program & **BEFORE** you run it. (you will get more information on the error by hovering over the error icon)

```
public class declareError {  
    public static void main(String[] args) {  
        int x;  
        int x;  
    }  
}
```

Problems @ Javadoc Declaration Console

<terminated> declareError [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 11, 2015, 2:49:24 PM)
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
Duplicate local variable x
at declareError.main(declareError.java:6)

The Java compiler even tells you the nature of your error (after you've tried to run the code)

Combining Declarations & Assignments

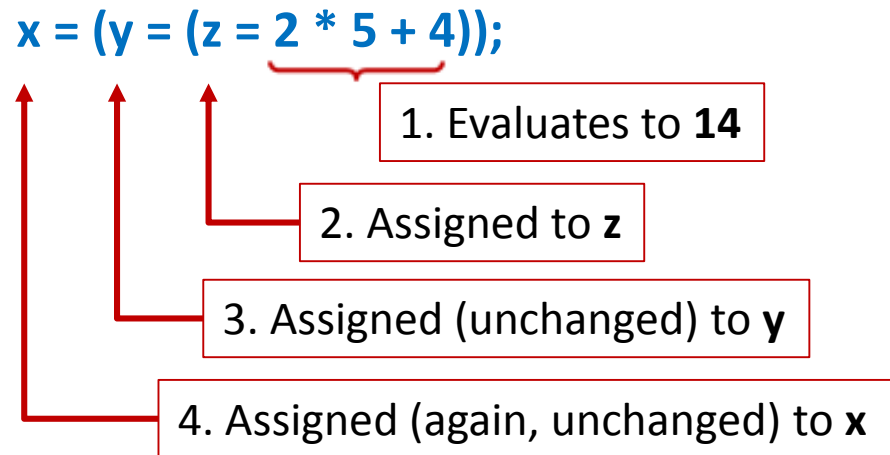
- The declaration and assignment statements can be combined into one line of code using the general syntax shown below:

<type> <name> = <expression>

- **Example:** `double weight = 120;`
`double height = 67;`
- More complex expressions may also be used: `double BMI = weight / (height * height) * 703;`
- Several variables of the same type may be declared at the same time: `int x, y, z;`
- Assignments may be mixed with declarations: `double height = 67, weight = 180, BMI;`

The assignment operator (=)

- The **KEY FEATURE** of an **assignment statement** is the **assignment operator (=)**
- The **assignment operator** evaluates **from RIGHT to LEFT**
- Thus, statements such as **x = y = z = 2 * 5 + 4;** are allowed in programs
- The example above is equivalent to the following:



- **End Result:** All 3 variables (x, y, z) have been assigned the value 14

```
int x, y, z;

x = y = z = 2 * 5 + 4;

System.out.println(x);
System.out.println(y);
System.out.println(z);
```

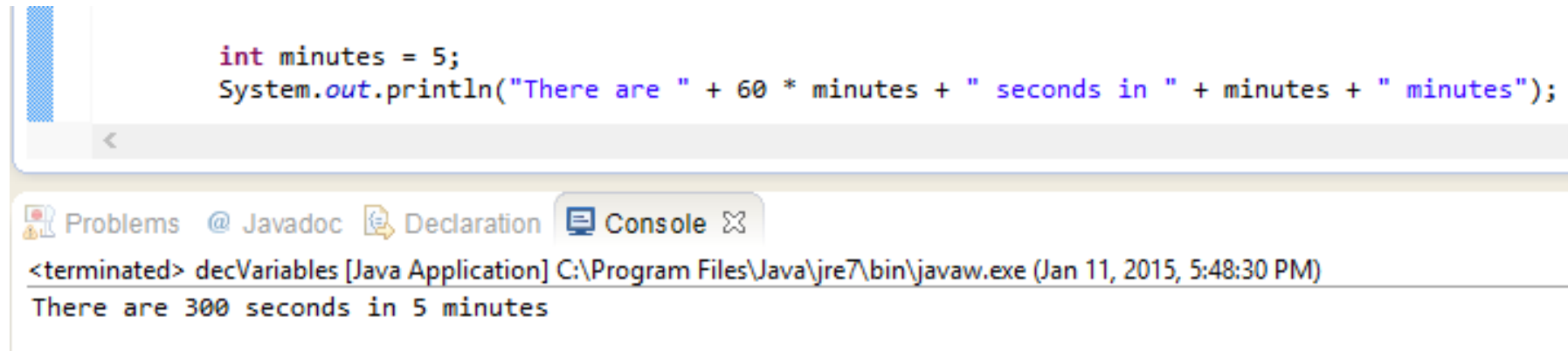
Problems @ Javadoc Declaration

<terminated> decVariables [Java Application] C

14
14
14

String Concatenation

- **String Concatenation:** Combining several strings into a single string, or combining a string with other data into a new, longer string
- String concatenation may be accomplished using the addition (+) operator
- The expression to be concatenated may include both strings (enclosed within quotes) and numbers, as well as variable names or expressions
- **Example:**



```
int minutes = 5;
System.out.println("There are " + 60 * minutes + " seconds in " + minutes + " minutes");
```

The screenshot shows an IDE window with a code editor and a console. The code editor contains the following Java code:

```
int minutes = 5;
System.out.println("There are " + 60 * minutes + " seconds in " + minutes + " minutes");
```

The console output shows the result of the program execution:

```
<terminated> decVariables [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 11, 2015, 5:48:30 PM)
There are 300 seconds in 5 minutes
```

Precedence in String Concatenation

- The “+” used in **string concatenation** has the SAME precedence as the “+” used for **arithmetic**
- This can lead to **CONFUSION!**
- Consider the expression: **2 + 3 + “hello” + 7 + 2 * 3** – How will the expression be evaluated by Java?

1. Multiplication operator has **HIGHEST** precedence

- After the highest precedence operator is addressed, we have: **2 + 3 + “hello” + 7 + 6**
- Now all the operators left in the expression have the **SAME** level of precedence
 - The evaluation of the expression will occur step-by-step from **LEFT** to **RIGHT**

2 + 3 + “hello” + 7 + 6

2. Both operations are integers, so Java interprets the “+” as the **addition operator** and evaluates this part of the expression to **5**

Precedence in String Concatenation, cont.

- We now have: $5 + \text{"hello"} + 7 + 6$
- At this point, the next operation would be between the **int** 5 and the **string** "hello"

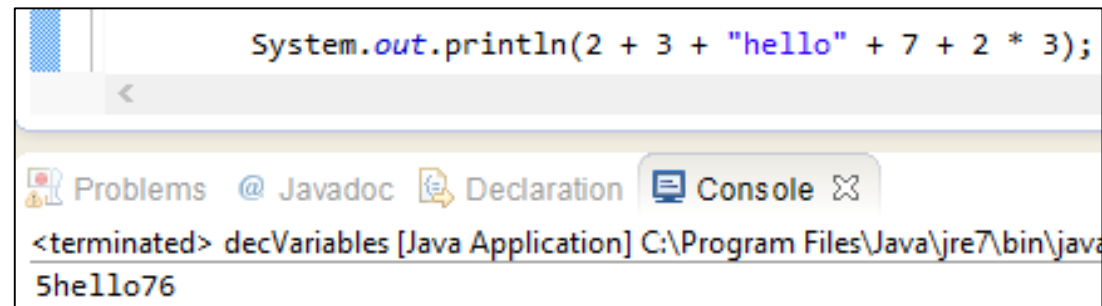
$5 + \text{"hello"} + 7 + 6$

3. Since this evaluation involves **mixed types**, Java converts the **int** to **string** type, so the evaluation essentially becomes "5" + "hello" and evaluates to "5hello"

- This gives us $\text{"5hello"} + 7 + 6$

4. The 2 remaining evaluations also involve **mixed types**

- **END RESULT:** "5hello76"



The screenshot shows a Java IDE window with a code editor and a console. The code editor contains the line: `System.out.println(2 + 3 + "hello" + 7 + 2 * 3);`. The console shows the output: `<terminated> decVariables [Java Application] C:\Program Files\Java\jre7\bin\java 5hello76`. The console output is split across two lines, with "5hello" on the first line and "76" on the second line.

Increment/Decrement Operators

- A frequent operation in programming is to change the value of a variable
 - Incrementing: Increasing the variable's value
 - Decrementing: Decreasing the variable's value
- Java provides special operators to facilitate such incrementing and decrementing

Desired Operation – Standard Operator	Concise Form Using Special Operator
a = a + 1;	a += 1;
b = b – 1;	b -= 1;
c = c + 2;	c += 2;
x = x * 2;	x *= 2;
y = y / 3;	y /= 3;

More Incrementing & Decrementing

- There is an even **more concise way** to express incrementing or decrementing the value of a variable by **1**

TO EXPRESS:	OPTION 1 (Post)	OPTION 2 (Pre)
<code>x = x + 1;</code>	<code>x++;</code>	<code>++x,</code>
<code>y = y - 1;</code>	<code>y--;</code>	<code>--y;</code>

- **Post-incrementing (x++) or post-decrementing (y--):**
 - Uses the special operator **AFTER** the variable
- **Pre-incrementing (++x) or pre-decrementing (--y):**
 - Uses the special operator **BEFORE** the variable

The difference between the two options becomes apparent when the special operators are **embedded** inside more complex expressions – a situation that should be handled with CAUTION.

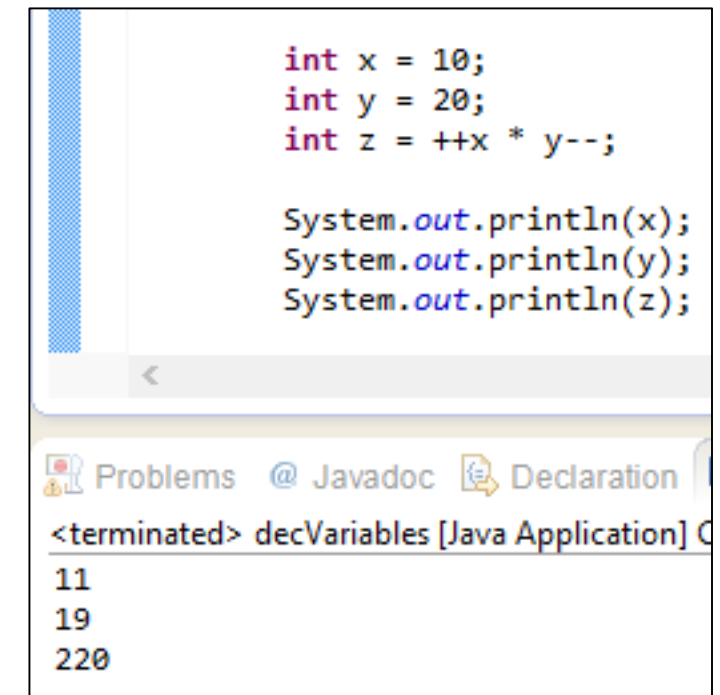
SEE NEXT SLIDE FOR AN EXAMPLE

Using pre/post increment/decrement operators inside complex expressions

- What would you expect the values of x, y, and z to be after the following lines of code are executed?

```
int x = 10;  
int y = 20;  
int z = ++x * y--;
```

- Because the first operand (**++x**) uses the **preincrement** operator, x is “updated” to **11 BEFORE** the entire expression is evaluated
- Because the second operand (**y--**) uses the **postincrement** operator, y is not “updated” until **AFTER** the entire expression is evaluated
- Thus **z** is evaluated as **11 * 20**
- After **z** is evaluated, **y** is updated to **19**, so the final results are:



The screenshot shows an IDE window with the following Java code:

```
int x = 10;  
int y = 20;  
int z = ++x * y--;  
  
System.out.println(x);  
System.out.println(y);  
System.out.println(z);
```

Below the code editor, the IDE's output console is visible, showing the results of the program execution:

```
<terminated> decVariables [Java Application] C  
11  
19  
220
```


Variables and Mixing Types

- As described, when you declare a variable in Java you **MUST** tell Java what type of data it will be storing
- What happens when you try to MIX types?
- In some cases, Java will allow the operation to occur
- The following statements are all legal – **except for the last one**:

```
// The following statements are all legal
int x = 2 + 3;
double y = 3.4 + 2.9;

// adding integers & storing the result in a double
double z = 3 + 4;

// The following is NOT legal!!
int a = 2.5 + 5.0;

}
```

This is legal because Java is able to “promote” the int to a double without losing any information

This is **NOT** legal because Java cannot convert a double to an int without the potential **LOSS** of information (any fractional part of the double will be lost)

The for Loop

- Loops help reduce redundancy in programs by repeatedly executing a sequence of tasks over a particular range of values
- The for loop is known as a “**definite loop**” because it executed a defined number of times
- Look at the following code examples:
 - Both code fragments produce the same output, but the loop does so much more compactly

```
// REDUNDANT WAY
System.out.println(1 + " squared = " + (1 * 1));
System.out.println(2 + " squared = " + (2 * 2));
System.out.println(3 + " squared = " + (3 * 3));
System.out.println(4 + " squared = " + (4 * 4));
System.out.println(5 + " squared = " + (5 * 5));
```

```
// USING A FOR LOOP
for(int i = 1; i <= 5; i++){
    System.out.println( i + " squared = " + (i * i));
}
```

OUTPUT

```
1 squared = 1
2 squared = 4
3 squared = 9
4 squared = 16
5 squared = 25
```

How the for Loop Works

- The general syntax of a for loop is as follows (compare to the example from the previous slide):

The loop header starts with the KEYWORD “for” and parentheses containing the following sections:

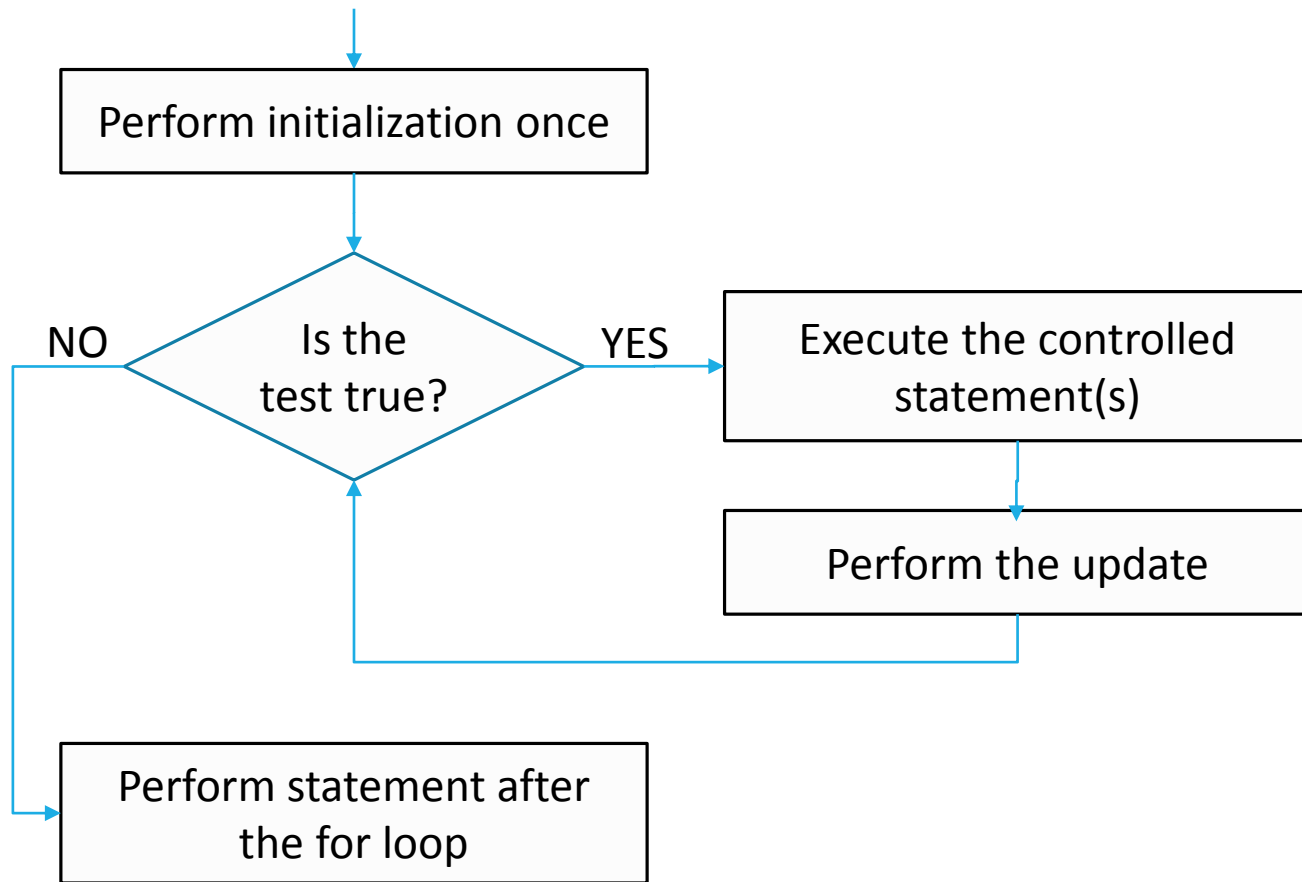
1. initialization: initializes a variable to serve as a counter
2. continuation test: performs a check to determine whether or not the loop execution should terminate
3. update: updates the variable serving as the loop counter

```
for (<initialization>; <continuation test>; <update>){  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}
```

The loop body contains the statements to be executed, contained within curly braces – these are the “controlled” statements

```
// USING A FOR LOOP  
for(int i = 1; i <= 5; i++){  
    System.out.println( i + " squared = " + (i * i));  
}
```

Flow of a for Loop



- **Control Structure:** A syntactic structure that controls other statements and, by doing so, affects the flow of control in a program
- Statements in a program generally proceed in sequential fashion, one after the other
- A loop interrupts this sequential flow, potentially causing the execution of the program to “double back on itself”, so that certain statements in the program may potentially be repeated

for Loop Patterns

- In our loop example, we set the loop counter to start at 1
- More commonly, this counter will start at 0 and execute n number of times
- Sometimes, it is more convenient to have a loop run “backwards” (e.g. to print a sequence of numbers in decreasing order)
 - In this case, the loop counter would be initialized to n and work backwards to some lower value
- The examples to the right show both forms of for loops:

CODE FRAGMENT:

```
// Incrementing Loop
for (int i = 0; i < 5; i++){
    System.out.print(i + " ");
}

System.out.println();
System.out.println("-----");

// Decrementing Loop
for (int i = 5; i > 0; i--){
    System.out.print(i + " ");
}
```

OUTPUT:

```
0 1 2 3 4
-----
5 4 3 2 1
```

Nested for Loops

- The **for loop** controls a statement, but the **for loop** itself is a statement
- This means that a **for loop** can control **ANOTHER** for loop
- When a loop is placed within another loop, it is referred to as a “**nested**” for loop
- **Example:**

```
// Nested Loop Demo
for (int i = 1; i <=3; i++){
    for (int j = 1; j <=3; j++){
        System.out.print(i + "." + j + " - ");
    }
    System.out.println();
}
```

OUTPUT:

```
1.1 - 1.2 - 1.3 -
2.1 - 2.2 - 2.3 -
3.1 - 3.2 - 3.3 -
```

Nested for Loop Example: Explanation

- What's going on in this example?

```
// Nested Loop Demo
for (int i = 1; i <=3; i++){
    for (int j = 1; j <=3; j++){
        System.out.print(i + "." + j + " - ");
    }
    System.out.println();
}
```

OUTPUT:

```
1.1 - 1.2 - 1.3 -
2.1 - 2.2 - 2.3 -
3.1 - 3.2 - 3.3 -
```

- The “inner loop” counter (j) executes 3 times for every iteration of the “outer loop”
- The outer loop counter executes 3 times (total)
- 3 iterations of the outer loop cause the controlled statement within the inner loop to execute a total of 3 X 3 (or 9) times
- In each number in the output, the value of the digit **before** the decimal represents the current value of **counter i**, while the value of the digit **following** the decimal represents the current value of **counter j**
- Thus, one iteration of the inner loop prints out the current value of the number represented by **i.j**

Scope of Variables

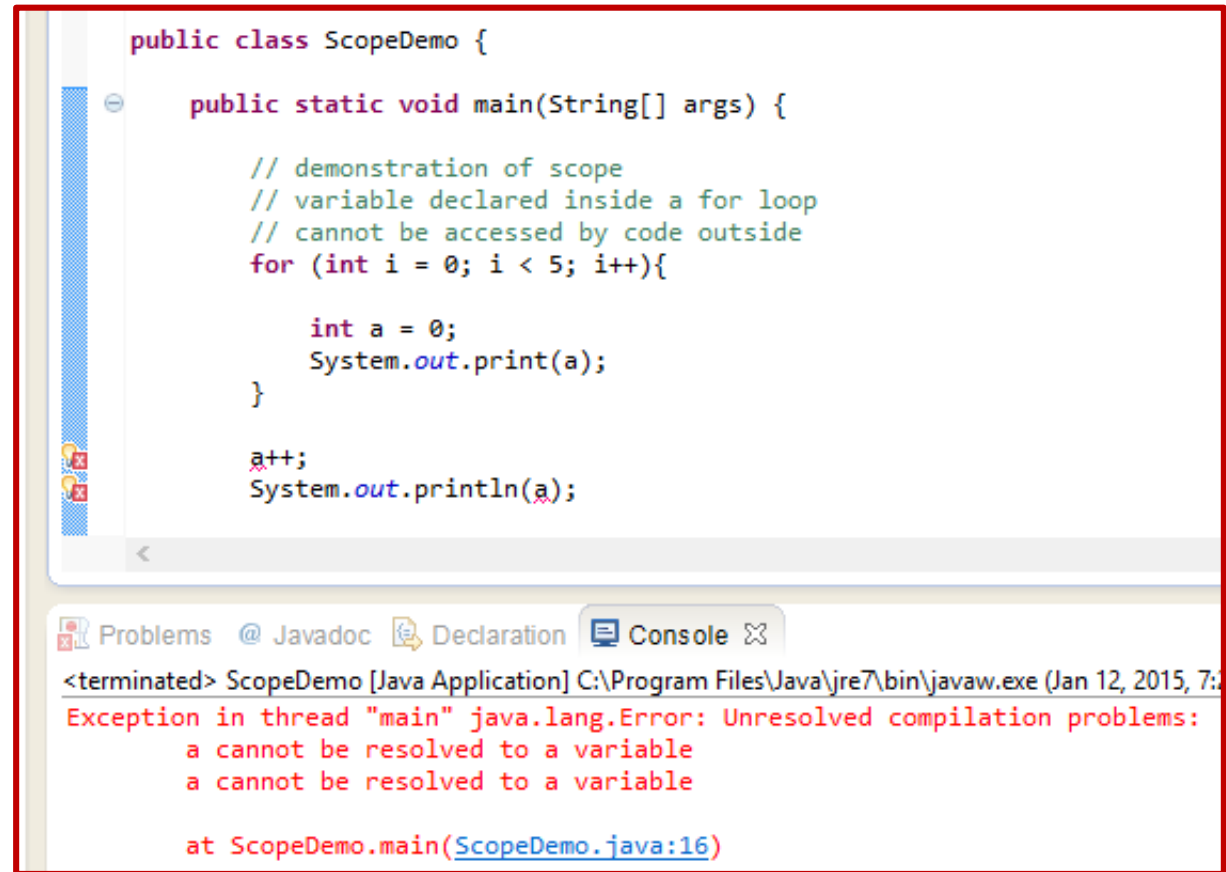
- While the scope of static methods is the entire class in which they appear, scope works differently for variables
- **Scope:** the part of a program in which a particular declaration is valid
- A variable is only valid in the block in which it is defined (i.e. statements grouped together by **curly braces**)
- **Local Variable**
 - Declared inside a method's curly braces
 - NOT accessible outside the method in which it was defined
- **Localizing Variables**
 - Declaring variables in the INNERMOST (most local) scope possible
 - Provides better SECURITY
 - Restricts access (thus preventing interference) from other parts of the program

Scope of Variables, cont.

- Look at the (rather silly) code shown below:

```
public class ScopeDemo {  
  
    public static void main(String[] args) {  
  
        // demonstration of scope  
        // variable declared inside a for loop  
        // cannot be accessed by code outside  
        for (int i = 0; i < 5; i++){  
  
            int a = 0;  
            System.out.print(a);  
  
        }  
  
    }  
}
```

- The variable **a** is only accessible to code **inside** the loop
- If code **outside** the loop tries to access the information stored in **a**, an **ERROR** results:



```
public class ScopeDemo {  
  
    public static void main(String[] args) {  
  
        // demonstration of scope  
        // variable declared inside a for loop  
        // cannot be accessed by code outside  
        for (int i = 0; i < 5; i++){  
  
            int a = 0;  
            System.out.print(a);  
  
        }  
  
        a++;  
        System.out.println(a);  
    }  
}
```

Problems @ Javadoc Declaration Console

<terminated> ScopeDemo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 12, 2015, 7:21:12 AM)
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
a cannot be resolved to a variable
a cannot be resolved to a variable
at ScopeDemo.main(ScopeDemo.java:16)

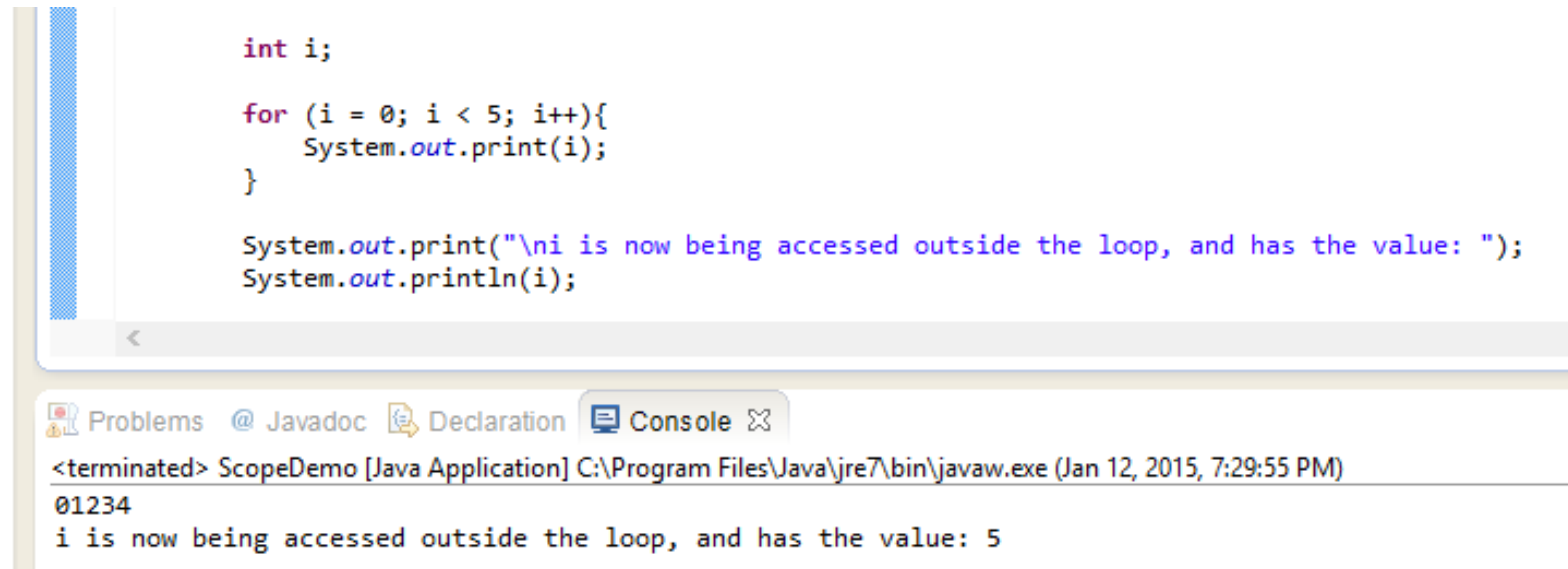
Scope of Variables, cont.

- The **loop control variable** is also local to the loop if it is declared in the loop header
- To access the loop control from outside the loop, you must declare it **before** using it in the loop header
- **Example:**

```
int i;

for (i = 0; i < 5; i++){
    System.out.print(i);
}

System.out.print("\ni is now being accessed outside the loop, and has the value: ");
System.out.println(i);
```



The screenshot shows an IDE window with a Java file. The code is as follows:

```
int i;

for (i = 0; i < 5; i++){
    System.out.print(i);
}

System.out.print("\ni is now being accessed outside the loop, and has the value: ");
System.out.println(i);
```

Below the code editor, the 'Console' tab is active, showing the output of the program:

```
<terminated> ScopeDemo [Java Application] C:\Program Files\Java\jre7\bin\javaw.exe (Jan 12, 2015, 7:29:55 PM)
01234
i is now being accessed outside the loop, and has the value: 5
```

- This can be useful if you want to have access to the final value of the loop control variable (or any other value modified during the execution of a loop)

Scope of Variables in Nested Loops

- In general, the same loop control variable should NOT be used in nested loops
- Each loop imposes its own termination and updating conditions, leading to likely interference between loops
- Consider the following cases:
 - In **CASE 1**, the outer loop control variable is **UPDATED** within the inner loop and becomes greater than the termination condition value during the **FIRST** iteration of the outer loop
 - In **CASE 2**, the inner loop **RESETS** the shared loop control variable during every iteration, so the variable's value **NEVER** satisfies the termination condition, creating what is known as an **INFINITE LOOP**

CASE 1:

```
int i;
for (i = 0; i < 5; i++){
    System.out.println(i);
    for (i = 0; i < 10; i++){
        System.out.print("-");
        System.out.print(i);
    }
}
```

Problems @ Javadoc Declaration Console
<terminated> ScopeDemo [Java Application] C:\Program Fil
0
-0-1-2-3-4-5-6-7-8-9

outer loop output

inner loop output

CASE 2:

```
int i;
for (i = 0; i < 10; i++){
    System.out.println();
    System.out.println(i);
    for (i = 0; i < 5; i++){
        System.out.print("-");
        System.out.print(i);
    }
}
```

Problems @ Javadoc Declaration Console
<terminated> ScopeDemo [Java Application] C:\Program Fil
-0-1-2-3-4
6
-0-1-2-3-4
6
-0-1-2-3-4
6
-0-1-2-3-4
6
-0-1-2-3-4
6
-0-1-2-3-4
6

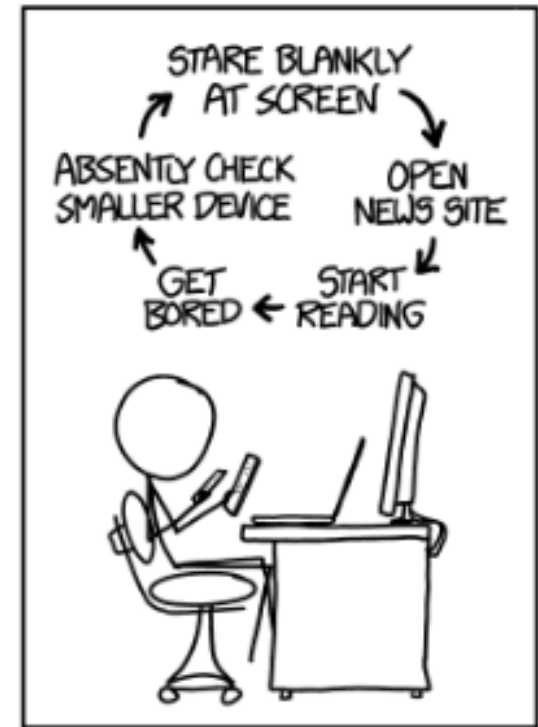
outer loop output

inner loop output

... and on and on "forever"

Infinite Loop

- A loop that **never terminates**, generally created due to:
 - Forgetting to update the loop control variable
 - Updating the wrong variable
 - A logic error



Cartoon from: <http://www.gizmodo.com.au/2014/08/the-modern-day-loop-of-infinite-self-distraction/>

Pseudocode

- English-like description of algorithms
- Should include enough detail to be easily translated into Java
- Consider the following instruction for drawing a box:
 - “ Draw a box with 5 lines and 3 columns of asterisks”
- Its language is too “informal” to provide any clues to how the appropriate Java code should be constructed
- **Helpful in development of a new program**
 - Lets you think through the logic of your solution without having to deal with the rigid structure and syntax of actual Java code

Pseudocode, cont.

BETTER:

```
for (each of 5 lines){  
    draw line of 10 asterisks  
}
```

- This method doesn't specify how to draw the line of output at the most basic possible level
- The second method neatly breaks the task into a lines/row and columns approach, which would be easier to generalize to create boxes of different dimensions

EVEN BETTER YET:

```
for (each of 5 lines){  
    for (each of 10 columns){  
        write 1 asterisk to output line  
    }  
}
```

- This is easy to translate to the following Java code:

```
for (int i = 0; i < 5; i++){  
    for(int j = 0; j < 10; j++){  
        System.out.print("*");  
    }  
    System.out.println();  
}
```

OUTPUT

```
<terminated>  
*****  
*****  
*****  
*****  
*****
```

Class Constants

- A class constant is a named value that cannot be changed
- It can be accessed from anywhere in the class (its scope is the entire class)
- You should choose descriptive names for class constants that describe what they represent
- For example, what if you wanted to figure out amounts of ingredients to buy to make a recipe?
 - How much you'll need to buy will depend on the number of servings you're making
 - This might vary each time to make the recipe
 - By using a class constant, you only need to change one value in your program
 - You could name this class constant **SERVINGS** (class constants are generally given UPPER CASE names)



Class Constants, cont.

- Constants are declared with the KEYWORD **final** – this indicates that their value **CANNOT** be changed at some later point in the program
- Constants can be declared anywhere variables can be declared
- Generally, constants are declared OUTSIDE methods, enabling them to be used by several different methods
- If you want static methods to have access to your constants, the constants must also be made static
- In addition, constants are generally made public
- General syntax: **public static final <type> <NAME> = expression;**
- **Example:**

```
public static final int SERVINGS = 6;
```


Concluding Comments

- 4 basic data types (“**primitives**”) commonly used in Java: **int**, **double**, **char**, and **boolean**
- Java **KEYWORDS** cannot be used as identifiers
- The simplest expressions are **LITERALS**
- Literals can be combined using operators to create **expressions**
- The common arithmetic operators are **+**, **-**, *****, **/**, and **%** (mod)
- **+** and **-** are of LOWER **precedence** than *****, **/**, and **%** (and thus get evaluated later)
- Mod (**%**) returns the **REMAINDER** of an **integer division**
- The basic unit of textual information is the **string literal**
- String literals can be **concatenated** using the **“+”** operator
- In **mixed expressions** containing digits and strings (e.g. **2 + “hello”**), the digit will be **converted** to a string

Concluding Comments, cont.

- **Variables** are locations in **memory** that store data
- Variables only “exist” within the blocks of code where they were created (this is their **scope**)
- **for** loops are used to **repeatedly** execute blocks of code
- Loops that never terminate are referred to as **INFINITE LOOPS**
- infinite loops are usually the result of a syntax or logic error
- **pseudocode** is useful in figuring out program logic before beginning actual coding and to explain the logic of your algorithms
- **class constants** hold values that cannot be changed by your program