

CSC3301 Java Workshop



Mohammed Hassan

mhassan.se@buk.edu.ng

Language Basics

Classes and Objects

Language Basics: Contents

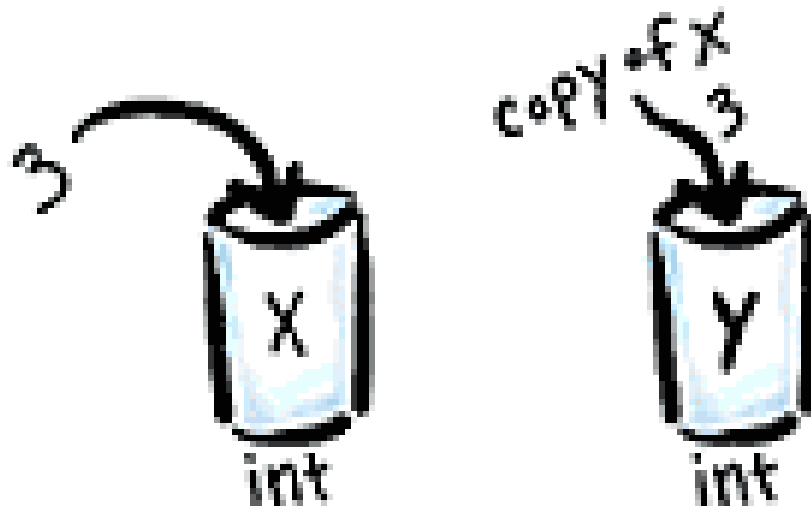
- ◆ Variables
- ◆ Operators
- ◆ Expressions, Statements, and Blocks
- ◆ Control Flow Statements

Variables

- ◆ You can imagine that a Java variable is a cup, with a value in it.
- ◆ What does it mean to say:

`int x = 3;`

`int y = x;`



Variables

- ◆ Variables in Java are very much like in C:

- `int` cadence; *// variable: type + name*
- `float` speed = 20.0f;
- `long` gear = 10L;

- ◆ Fields (or attributes) are variables that are used by an **object** to store its state:

```
class AA {  
    int field = 4;            // an instance variable  
    static char c = 'a'; // a class variable is shared  
}
```

Kinds of Variables

- ◆ **Instance variables** (or non-static fields) are used by an object to store its state.

```
class Bicycle {           // fields or attributes
    int cadence = 0; // instance variables
    int speed = 0;
    int gear = 1;
```

...

See the Implementation of a Bicycle, Lecture 1

- ◆ **Class variables** (or static fields) – there is **only 1** copy per class, i.e., all the objects share that class variable (i.e., static field).

```
class Bicycle {           // fields or attributes
    int cadence = 0;
    int speed = 0;
    int gear = 1;
    static int numGears = 6; // class variable
```

...

Kinds of Variables

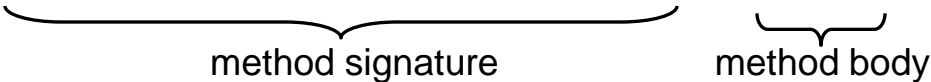
- ◆ **Local variables** are used by methods to store temporary values.

```
void method( ... ) {  
    int localVariable = 0;
```

```
    ...  
}
```

- ◆ **Parameters** are the variables passed to a method.

```
void method(int parameter) { ... }
```



```
void changeGear(int newValue) { // See the Bicycle class,  
                                // Lecture 1,
```

```
    gear = newValue;
```

```
}
```

Naming Conventions

Identifier Type	Rules for Naming	Examples
Variables	Variable names are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore _ or dollar sign \$ characters.	<code>char c;</code> <code>float myWidth;</code>
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_").	<code>static final int MAX_WIDTH = 999;</code> <code>static final int GET_THE_CPU = 1;</code>
Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	<code>run();</code> <code>runFast();</code>

Naming Conventions

Identifier Type	Rules for Naming	Examples
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized.	class Raster ; class ImageSprite ;
Interfaces	Interface names should be capitalized like class names.	interface RasterDelegate ; interface Storing;
Packages	The prefix of a unique package name is always written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries. Subsequent components of the package name vary according to an organization's own internal naming conventions.	com.sun.ng edu.cmu.cs.bovik.cheese

Primitive Data Types

- ◆ Java is a strongly typed language:
 - All variables must be defined before use;
 - The variable's type name must be stated.
- ◆ The compiler assigns a *default value* to an *uninitialized field*.
- ◆ The compiler never assigns a default value to an uninitialized local variable.
- ◆ Using an uninitialized local variable will result in a **compile-time error**.

Primitive Type	Definition	Default Value for Fields
boolean	either <i>true</i> or <i>false</i>	false
byte	8-bit signed integer	0
char	16-bit Unicode UTF-16 character	'u0000'
short	16-bit signed integer	0
int	32-bit signed integer	0
long	64-bit signed integer	0L
float	32-bit signed floating point	0.0F
double	64-bit signed floating point	0.0D

Character Strings

- ◆ Java provides special support for character strings via the ***String*** class.
- ◆ A String is an immutable sequence of characters (it cannot be changed after it is created):
 - `String s1 = new String("this is a String");`
 - `String s2 = "this is another String";`
 - `String s2 = null; // no String object assigned`
- ◆ The String class is defined in the *java.lang* package, i.e., *java.lang.String*.

Literals

- ◆ A literal is the source code representation of a fixed value.
- ◆ Literals do not require computation.
- ◆ Java supports special escape sequences for char and String literals:
 - \b – backspace
 - \t – tab
 - \n – line feed
 - \f – form feed
 - \r – carriage return
 - \" – double quote
 - \' – single quote
 - \\ – backslash

```
boolean result = true;  
char capitalC = 'C';
```

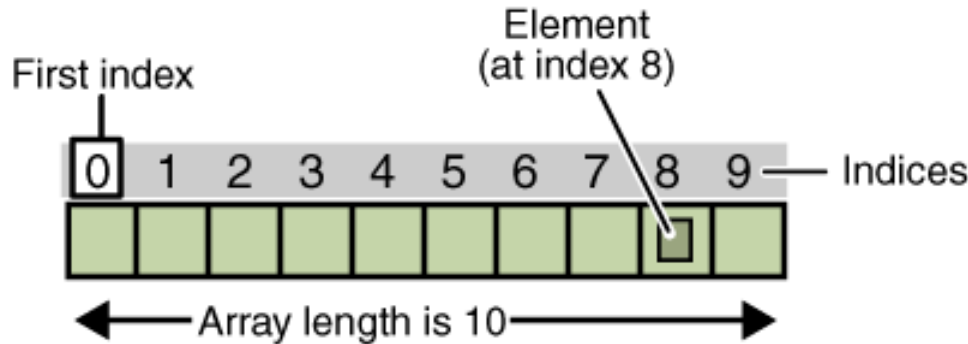
```
int decVal = 26;  
int octVal = 032;  
int hexVal = 0x1a;
```

```
double d1 = 123.4;  
double d2 = 1.234e2;  
float fl = 123.4f;
```

- ◆ *null* is a special literal that can be assigned to any variable that is **not** a primitive type:

```
String s2 = null;  
byte b = null; // error
```

An array of ten elements



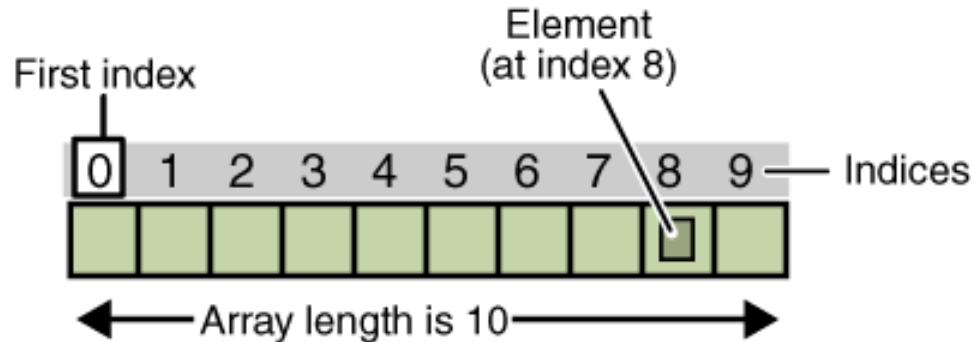
```
int[] anArray; // declares an array of integers
anArray = new int[10]; // allocates memory for 10 integers
```

OR

```
int[] anArray = new int[10]; // declaring and allocating a
// memory of ten integers to an
// array of integers
```

```
anArray[0] = 100; // initialize first element
anArray[1] = 200; // initialize second element
// prints values of the first and the second element
System.out.println("Element at index 0: " + anArray[0]);
System.out.println("Element at index 1: " + anArray[1]);
```

An array of ten elements

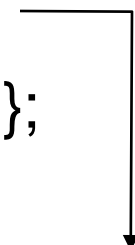


```
class ArrayDemo {  
    public static void main(String[] args) {  
        int[] anArray;           // declares an array of integers  
        anArray = new int[10];   // allocates memory for 10 integers  
  
        anArray[0] = 100; // initialize first element  
        anArray[1] = 200; // initialize second element  
        System.out.println("Element at index 0: " + anArray[0]);  
        System.out.println("Element at index 1: " + anArray[1]);  
        // prints values of the first and the second element  
    }  
}
```

Arrays

- ◆ An array is a container that holds a fixed number of values of a single type.
- ◆ The length of an array is defined upon its creation, and it cannot be changed.
- ◆ Each item in an array is called an *element*.
- ◆ Each element is accessed by its numerical *index* (from 0 to *length-1*).

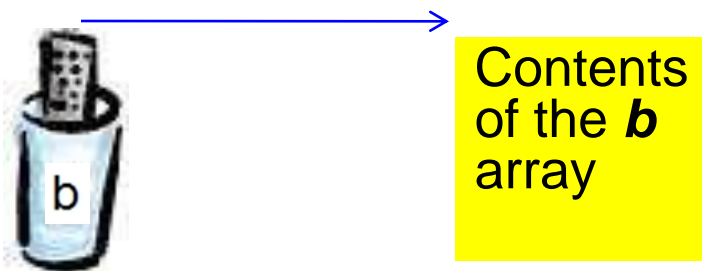
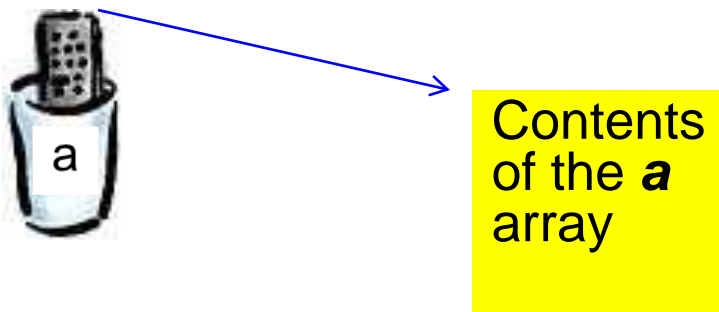
```
int[] a1 = new int[5];  
int[] a2 = { 1,2,3,4,5 };  
  
int aL = a1.length // = 5  
a2.length = 6;      // error  
  
a1[0] = 1;  
a2[a2.length-1] = 5;  
a1[5] = 6;           // error  
a2[-1] = -1;         // error
```



Arrays

◆ Two arrays:

- `int[] a = {1, 2, 3, 4, 5};`
- `int[] b = {15, 16, 17, 18, 19};`



- ## ◆ A REFERENCE to the array is stored as a value of the variable (in the cup, as shown in the picture). Think of it as a remote control to the contents of the array.

Arrays of Objects

- ◆ Java supports arrays of objects.

```
String[] a1 = new String[5];  
String[] a2 = { "1", "2" };  
String[] a3 = { new String("1"), "2" };
```

- ◆ The elements/objects in an array must belong to the same type/class.

```
a1[1] = "str";  
a2[0] = a1[1];  
a1[0] = 444; // error
```

- ◆ An array can be printed out one element at a time.

```
System.out.println(a3[0]);  
System.out.println(a3[1]);
```


Multidimensional Arrays

	Column 0	Column 1	Column 2
Row 0	<code>x[0][0]</code>	<code>x[0][1]</code>	<code>x[0][2]</code>
Row 1	<code>x[1][0]</code>	<code>x[1][1]</code>	<code>x[1][2]</code>
Row 2	<code>x[2][0]</code>	<code>x[2][1]</code>	<code>x[2][2]</code>

- ◆ A multidimensional array is simply an array whose components are themselves arrays.
- ◆ This is unlike arrays in C or Fortran. A consequence of this is that the rows are allowed to vary in length (ragged arrays).

Multidimensional Arrays

```
String[][] names;           // declares an array of type sting
names = new String[3][4]; //allocate memory of 3 rows
                           //and 4columns
```

OR

```
String[][] names = new String[3][4]; //declaration and memory
                                     //size allocation
```

```
// initialize the elements
```

```
names[0][0] = "Mr";
names[0][1] = "Mrs";
names[1][0] = "Hassan";
....
```

Multidimensional Arrays: Example

```
class MultiDimArrayDemo {  
    public static void main(String[] args) {  
        String[][] names = {"Mr. ", "Mrs. ", "Ms. ",  
                             {"Smith", "Jones"}};  
        System.out.println(names[0][0] + names[1][0]); //Mr. Smith  
        System.out.println(names[0][2] + names[1][1]); //Ms. Jones  
    }  
}
```

Summary of Variables

- ◆ The term ***instance variable*** is another name for a **non-static** field (or attribute).
- ◆ The term ***class variable*** is another name for a **static** field.
- ◆ A **local variable** is declared inside a method. It stores temporary state.
- ◆ A ***parameter*** is a variable declared within the parentheses of a method signature.
- ◆ The 8 primitive (or *native*) data types are: *byte*, *char*, *short*, *int*, *long*, *float*, *double*, and *boolean*.
- ◆ Character strings are represented by the class **String**
- ◆ An array is a ***container*** of *object* that holds a fixed number of values of a **single** type.
- ◆ ***null*** is the only literal object reference. It represents an invalid object or one that has not been created yet.

Operator Precedence

high↑

Operator	Precedence	
postfix	expr++ expr--	
unary	++expr --expr +expr -expr ~ !	
multiplicative	* / %	
additive	+ -	
shift	<< >> >>>	
relational	< > <= >= instanceof	
equality	== !=	
bitwise AND	&	
bitwise exclusive OR	^	
bitwise inclusive OR		
logical AND	&&	
logical OR		
ternary	? :	
assignment	= += -= *= /= %= &= ^= = <<= >>= >>>=	

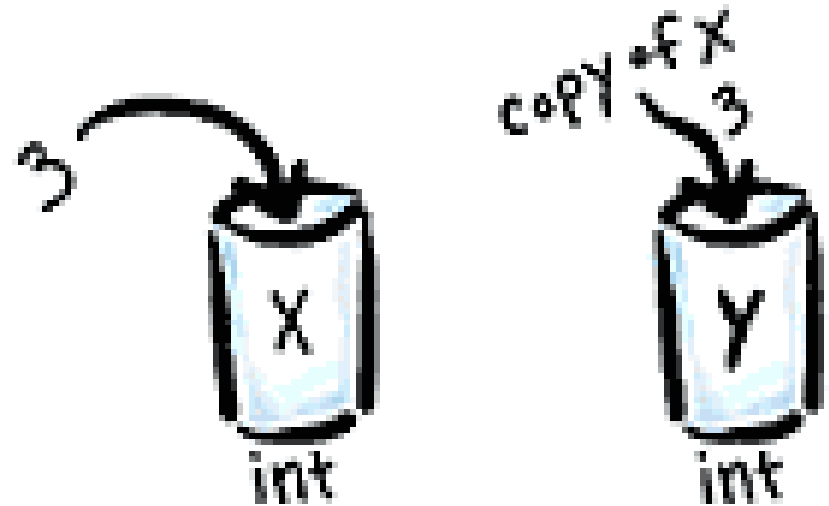
low

The Assignment Operator

- ◆ The most common operator is the assignment operator '='

```
int x = 3;
```

```
int y = x;
```



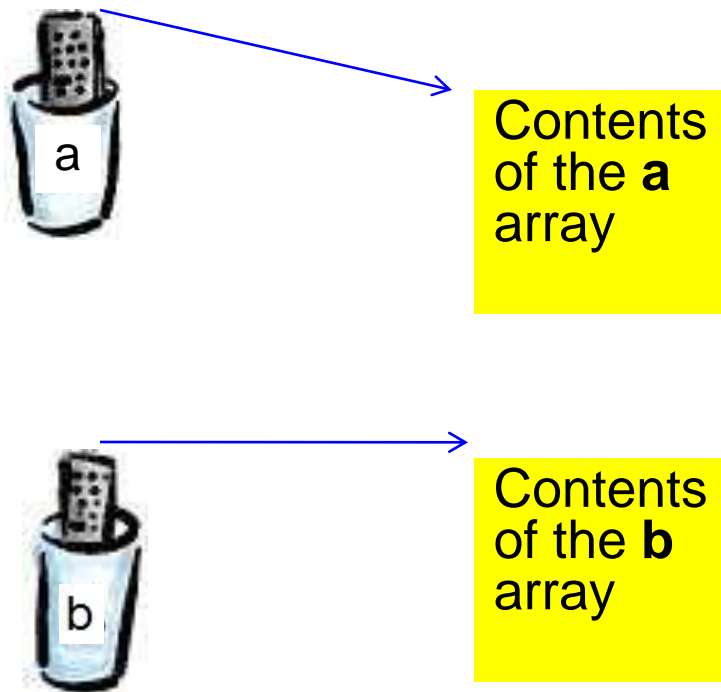
```
boolean b = true;
```

```
float speed = 120.0f;
```

Copying Arrays

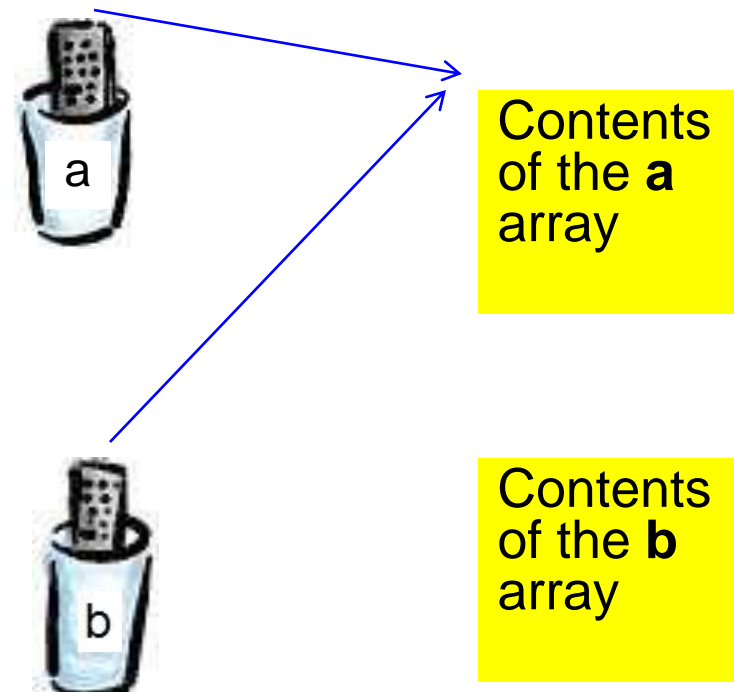
- ◆ Two arrays:

- `int[] a = {1, 2, 3, 4, 5};`
- `int[] b = {15, 16, 17, 18, 19};`



- ◆ The picture below:
After the assignment

- `b = a;`



- ◆ The picture above:
Before assignment

- `b = a;`

Copying Arrays

- ◆ The *System* class has an *arraycopy* method that you can use to copy data from one array into another:

```
class ArrayCopyDemo {  
    public static void main(String[] args) {  
        char[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',  
                             'i', 'n', 'a', 't', 'e', 'd' };  
        char[] copyTo = new char[7];  
  
        System.arraycopy(copyFrom, 2, copyTo, 0, 7);  
        System.out.println(new String(copyTo));  
    }  
}
```

The output from this program is:

caffeine

The Arithmetic Operators

- ◆ Java provides operators that perform:
 - Addition: `+` the additive operator
 - Subtraction: `-` the subtraction operator
 - Multiplication: `*` the multiplication operator
 - Division: `/` the division operator
 - Remainder: `%` the remainder operator
- ◆ Examples:
 - `int result = 1 + 2;` // → `result = 3`
 - `int result = 13 % 2;` // → `result = 1`

Concatenating Two Strings

- ◆ The + operator can be used to join or concatenate two strings:

```
String s1 = new String("aa");
```

```
String s2 = "bb";
```

```
String s3 = s1 + s2;
```

```
System.out.println(s3);    // → "aabb"
```

The Unary Operators

The unary operators require only 1 operand:

- + unary plus operator, indicates positive value
- - unary minus operator, indicates negative value
- ++ increment operator, increments a value by 1
- -- decrement operator, decrements a value by 1
- ! logical complement operator, inverts the value of a boolean

Examples:

```
int result = -1;
```

```
result++;           // (postfix)    → result = 0
```

```
--result;          // (prefix)    → result = -1
```

```
++result;           //              → result = 0
```

```
boolean b = true;
```

```
b = !b;             // → b = false
```

The Equality and Relational Operators

◆ The equality and relational operators are:

- == equal to
- != not equal to
- > greater than
- >= greater than or equal
- < less than
- <= less than or equal to

◆ Examples:

```
int m;
```

```
if (1 == 2) { m = 1; } else { m = 3; } // → false; m received 3
```

```
if (2 > 1) {m = 5;} // → true; m received 5
```

```
int value = 1;
```

```
if (value != 0) { m = 7; } else { m = 9; } // m received 7
```

The Conditional Operators

◆ The conditional operators are:

- && conditional AND
- || conditional OR
- ? : ternary operator

◆ Examples:

```
int m = 5;
```

```
boolean b1 = true, b2 = false;
```

```
if (b1 && b2) { m = 10; } // → false; the value of m is 5
```

```
if (b1 || b2) { m = 15; } // → true; m received 15
```

```
boolean b = 1 > 0 ? true : false; // → b = true
```

Type Comparison Operator

- ◆ The *instanceof* operator compares an object to a specified class.
- ◆ *instanceof* is used to test if an object is an instance of a class or a subclass, or an instance of (a class that implements) an interface.
- ◆ An example:

```
String str = new String("123");  
if (str instanceof String) {           // → true  
    System.out.println("The type of str is String");  
}
```

Bitwise and Bit Shift Operators

- ◆ These operators operate on integral types.
- ◆ The bitwise operators are:
 - `&` bitwise AND
 - `^` bitwise XOR – exclusive OR
 - `|` bitwise OR
 - `~` the complement operator
- ◆ For example:
 - `0010 & 0110 // → 0010`
 - `0010 | 0110 // → 0110`
 - `0010 ^ 0110 // → 0100`
 - `~0100 // → 1011`
- ◆ The bit shift operators shift bits left or right.
- ◆ The signed bit shift operators are:
 - `<<` shifts to the left
 - `>>` shifts to the right
- ◆ The unsigned bit shift operator is:
 - `>>>` shifts to the right and fills with 0 bits on the left
- ◆ For example:
 - `0001 << 1 // → 0010`
 - `0010 >> 1 // → 0001`
 - `1001 >>> 1 // → 0100`

Expressions and Statements

- ◆ An expression is a construct that consists of variables, operators, and method invocations.
- ◆ Examples are in a blue color below:
`int a = 1;`
`int b = 2;`
`int c = a * b + 3;`
- ◆ Statements are equivalent to sentences in natural languages. A statement forms a complete unit of execution.
- ◆ Examples:
`aValue = 4;`
`Car c = new Car();`
`double db = 4.;`

Blocks

- ◆ A block is a group of zero or more statements between balanced braces.
- ◆ Blocks can be used anywhere a single statement is allowed.

```
class BlockDemo {  
    public static void main(String[] args) {  
        boolean condition = true;  
        if (condition) { // begin block 1  
            System.out.println("Condition is true.");  
        } // end block one  
        else { // begin block 2  
            System.out.println("Condition is false.");  
        } // end block 2  
    }  
}
```

Summary of Operators

- ◆ Operators may be used to build expressions that compute values.
- ◆ Expressions are the core components of statements.
- ◆ Statements may be grouped into blocks.
- ◆ Statements end with a semicolon ‘;’
- ◆ A block is a group of zero or more statements between balanced braces ‘{’ and ‘}’.
- ◆ Blocks can be used anywhere a single statement is allowed.

Control Flow Statements

- ◆ The statements inside a Java source file are generally executed from top to bottom, in the order that they appear.
- ◆ Control flow statements break up the flow of execution via:
 - Decision making – *if*, *if-else*, *switch*
 - Looping – *for*, *while*, *do-while*
 - Branching – *break*, *continue*, *return*

The *if-then* Statement

- ◆ The *if-then* statement instructs the computer to execute a certain section of code only if a particular test evaluates to true.
- ◆ An example:

```
int a = 4;
```

```
int c = 9;
```

```
if (a < 5) {
```

```
    a++;
```

```
    c = a + 4;
```

```
}
```



The *if-else* Statement

- ◆ The *if-else* statement provides a secondary path of execution when an *if* clause evaluates to false.
- ◆ For example:

```
if (a < 5) {  
    ...  
}  
else {  
    ...  
}
```

if the expression is true

if the expression is false

Multiple 'else if' blocks

- ◆ An example: Assigning a grade based on the value of testscore

```
class IfElseDemo {  
    public static void main(String[] args) {  
        int testscore = 76;  
        char grade;  
        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("Grade = " + grade); // Output: Grade =C  
    }  
}
```

The *switch* Statement

- ◆ The switch statement allows any number of possible execution paths.
- ◆ A switch works with the following data types: *byte*, *short*, *char*, and *int*.
- ◆ A switch works with some other types (e.g., Integer, Short, enumerated types, etc.):
 - Integer is a wrapper class for the type int
 - Short is a wrapper class for the type short

```
final int month = 2;
String name;
switch (month) {
    case 1:
        name = "january";
        break;
    case 2:
        name = "february";
        break;
    default:
        name = "";
        break;
}
System.out.println(name);
// output: february
```

Example: switch

```
final Integer month = 4;  
String name;
```

```
switch (month) {  
    case 1:  
        name = "january";  
        break;  
    case 2:  
        name = "february";  
        break;  
    default:  
        name = "";  
        break;  
}  
System.out.println(name);  
// output: an empty string
```

```
Short month = new Short(2);  
String name;
```

```
switch (month) {  
    case 1:  
        name = "january";  
        break;  
    case 2:  
        name = "february";  
        break;  
    default:  
        name = "";  
        break;  
}  
System.out.println(name);  
// output february
```


Example: switch And if-else if

```
int month = 10;  
String name;  
switch (month) {  
    case 1:   
        name = "january";  
        break;  
    case 2:   
        name = "february";  
        break;  
    default:  
        name = "";  
        break;  
}
```

jumps directly

The diagram illustrates the execution of a switch statement. Dashed arrows show the flow from the 'switch' line to 'case 1:', then to 'case 2:', and finally to 'default:'. Another dashed arrow points from the 'break;' statement in the 'case 1:' block back to the 'switch' line, indicating a direct jump to the next case or the end of the switch block.

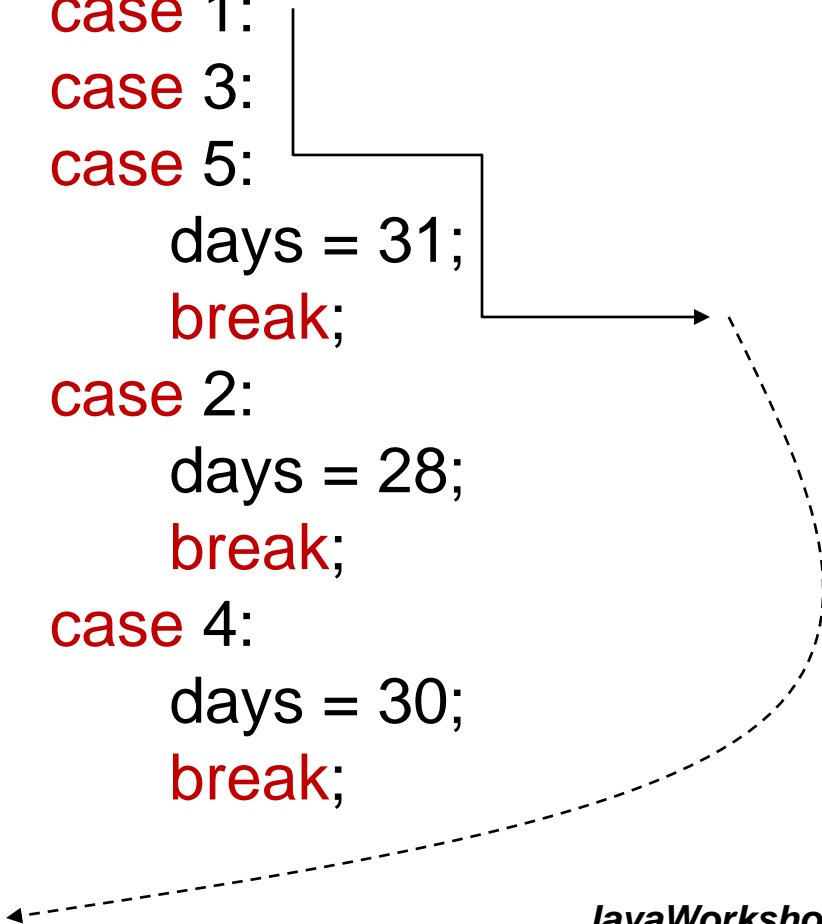
```
final int month = 10;  
String name;  
if (month == 1)  
    name = "january";  
else if (month == 2)  
    name = "february";  
else  
    name = "";  
System.out.println(name);
```

checks one by one

The diagram illustrates the execution of an if-else if-else statement. A vertical dashed arrow points downwards from the 'if' line to the 'else if' line, and then to the 'else' line, indicating that each condition is checked sequentially until one is true.

Example: A *fall-through* switch

```
int month = 1;  
switch (month) {  
    case 1:  
    case 3:  
    case 5:  
        days = 31;  
        break;  
    case 2:  
        days = 28;  
        break;  
    case 4:  
        days = 30;  
        break;  
}
```



The *while* Statement

- ◆ The *while* loop continually executes a block of statements as long as a particular condition is true:

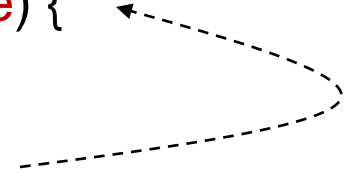
```
while (condition is true) {  
    ...  
}
```

- ◆ An example:

```
int i = 1;  
while (i < 5) {  
    System.out.println(i++);  
}
```

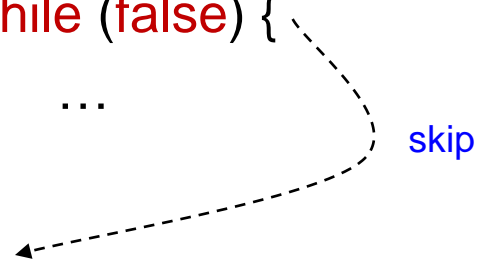
- ◆ An infinite loop as a *while* block:

```
// loops forever  
while (true) {  
    ...  
}
```

A dashed arrow originates from the closing curly brace of the while loop and points back to the opening curly brace, indicating a continuous cycle.

- ◆ This loop never runs:

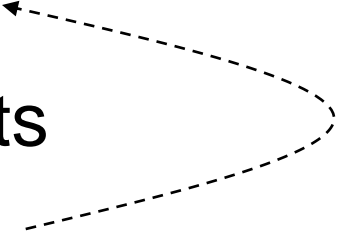
```
while (false) {  
    ...  
}
```

A dashed arrow originates from the closing curly brace of the while loop and points away from the loop structure to the right, labeled with the word "skip" in blue, indicating that the loop body is bypassed because the condition is false.

The *do-while* Statement

- ◆ The *do-while* loop checks its *condition* of termination after its block has executed:

```
do {  
    ...    // statements  
} while (condition);
```



loops while the condition is true

- ◆ A *do-while* loop executes at least once
- ◆ A *while* loop may or may not execute

Example: do-while

◆ Correct:

```
int[] array = new int[2];  
int i = 0;
```

```
do {  
    array[i] = i;  
    ++i;  
} while (i < array.length);
```

◆ Incorrect:

```
int[] array = new int[2];  
int i = array.length;
```

```
do {  
    array[i] = i; // error: i = 2  
    --i;  
} while (i >= 0);
```

The *for* Statement

- ◆ The *for* loop executes repeatedly until a termination condition is not satisfied:

```
for (initialization ; condition_of_termination ; increment) {  
    ...  
}
```
- ◆ For example:

```
for (int i = 0; i < 10; ++i) {  
    System.out.println(i); // prints 10 lines  
}
```
- ◆ An infinite loop can be expressed as:

```
for (;;) {  
    ...  
}
```

Example: for And do-while

```
int[] array = new int[2];  
int i = array.length;
```

```
for (;i < array.length;) {  
    array[i] = i;  
    ++i;  
}
```

skip

```
int[] array = new int[2];  
int i = array.length;
```

```
do {  
    --i;  
    array[i] = i;  
} while (i < array.length);
```

no skip

The *break* Statements

- ◆ The *break* statement has two forms: labeled, and unlabeled
- ◆ An unlabeled break can be used to terminate a for, while, or do-while loop, and a switch
- ◆ A labeled break statement terminates an outer statement

```
int i = 0;  
while (true)  
    if (i > 5)
```

```
        break;
```

```
    else
```

```
        ++i;
```



```
labeled_break:
```

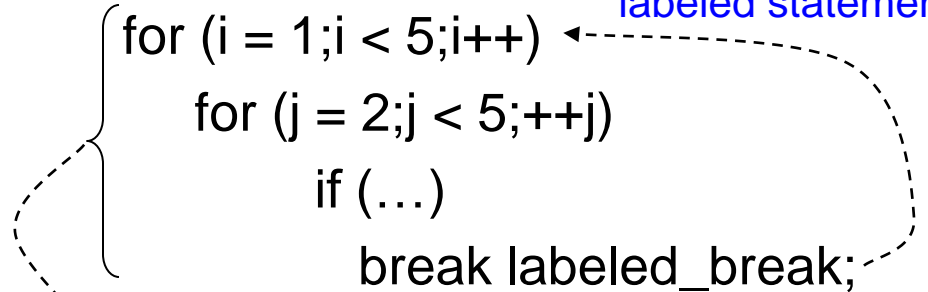
```
{ for (i = 1; i < 5; i++)
```

```
    for (j = 2; j < 5; ++j)
```

```
        if (...)
```

```
            break labeled_break;
```

1. terminates the
labeled statement

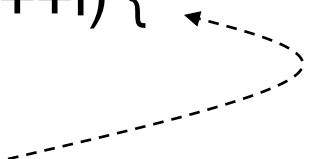


2. skips the block upon termination

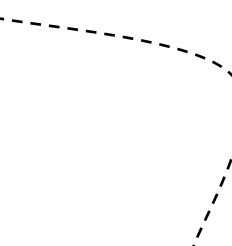
The *continue* Statement

- ◆ It skips the current iteration of a for, while, or do-while loop.
- ◆ The unlabeled form skips to the end of the innermost loop's body.
- ◆ A labeled continue statement skips the current iteration of an outer loop marked with the given label.

```
for (int i = 1; i < 10; ++i) {  
    if (i > 5)  
        continue;  
    System.out.println(i);  
}
```



```
label :  
for (int i = 1; i < 10; ++i) {  
    for (int j = 0; j < 5; j++)  
        if (i > 5)  
            continue label;  
    System.out.println(i);  
}
```



The *return* Statement

- ◆ The return statement exits from the current method, and control flow returns to where the method was invoked.

```
void method1()  
{  
    int i = method2();  
    return; // no return value  
}
```

- ◆ A return statement may or may not return a value, for example:

```
return;  
return 5;
```

```
int method2()  
{  
    int i = 0;  
    i += 5;  
    return i; // must return an int  
}
```

Summary of Control Flow Statements

- ◆ The *if-then* statement tells your program to execute a certain section of code *only if* a particular test evaluates to true.
- ◆ The *if-then-else* statement provides a secondary path of execution when an "if" clause evaluates to false.
- ◆ The *switch* statement allows for any number of possible execution paths.
- ◆ The *while* and *do-while* statements continually execute a block of statements while a particular condition is true.
- ◆ The *for* statement provides a compact way to iterate over a range of values.

Lab Exercise: Basic Operations on Triangular Matrices

- ◆ You should know that a matrix of the form:

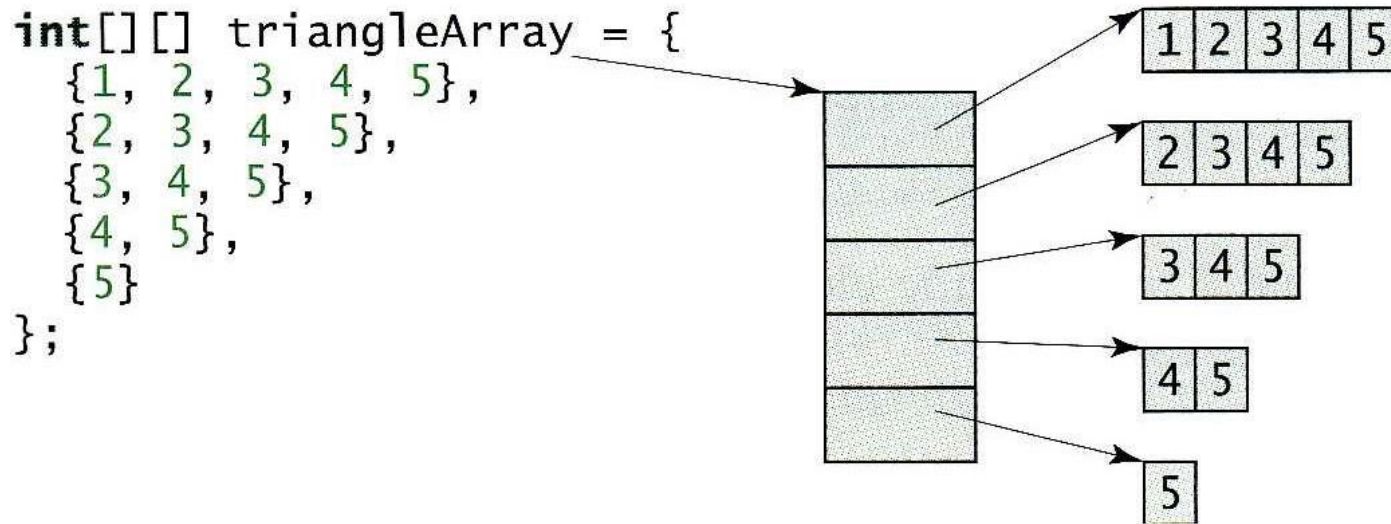
$$\mathbf{L} = \begin{bmatrix} l_{1,1} & & & & 0 \\ l_{2,1} & l_{2,2} & & & \\ l_{3,1} & l_{3,2} & \ddots & & \\ \vdots & \vdots & \ddots & \ddots & \\ l_{n,1} & l_{n,2} & \dots & l_{n,n-1} & l_{n,n} \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & \dots & u_{1,n} \\ & u_{2,2} & u_{2,3} & \dots & u_{2,n} \\ & & \ddots & \ddots & \vdots \\ & & & \ddots & u_{n-1,n} \\ 0 & & & & u_{n,n} \end{bmatrix}$$

L is called lower triangular matrix or left triangular matrix, and similar matrix U is called Upper triangular matrix or right triangular matrix

- ◆ The standard operations on triangular matrices conveniently preserve the triangular form: the sum and product of two upper triangular matrices is upper triangular. The inverse of an upper triangular matrix is also upper triangular. The result of the multiplication of an upper triangular matrix by a constant is the upper triangular matrix. The analogous result holds for lower triangular matrices. Because matrix equations with triangular matrices are easier to solve they are very important in numerical analysis. Triangular matrices can be presented by ragged arrays (See lecture slides). An example is in the next slide.

Lab Exercise: Basic Operations on Triangular Matrices



- ◆ Your program should find the sum of an upper triangular matrix and a lower triangular matrix. The result should be printed on the screen. You should use ragged arrays for triangular matrices in your Java code. The number of rows in your matrices should be equal to 6. You make decision about the type of elements and their values. The name of your source file is *SumTriangularMatrices.java*