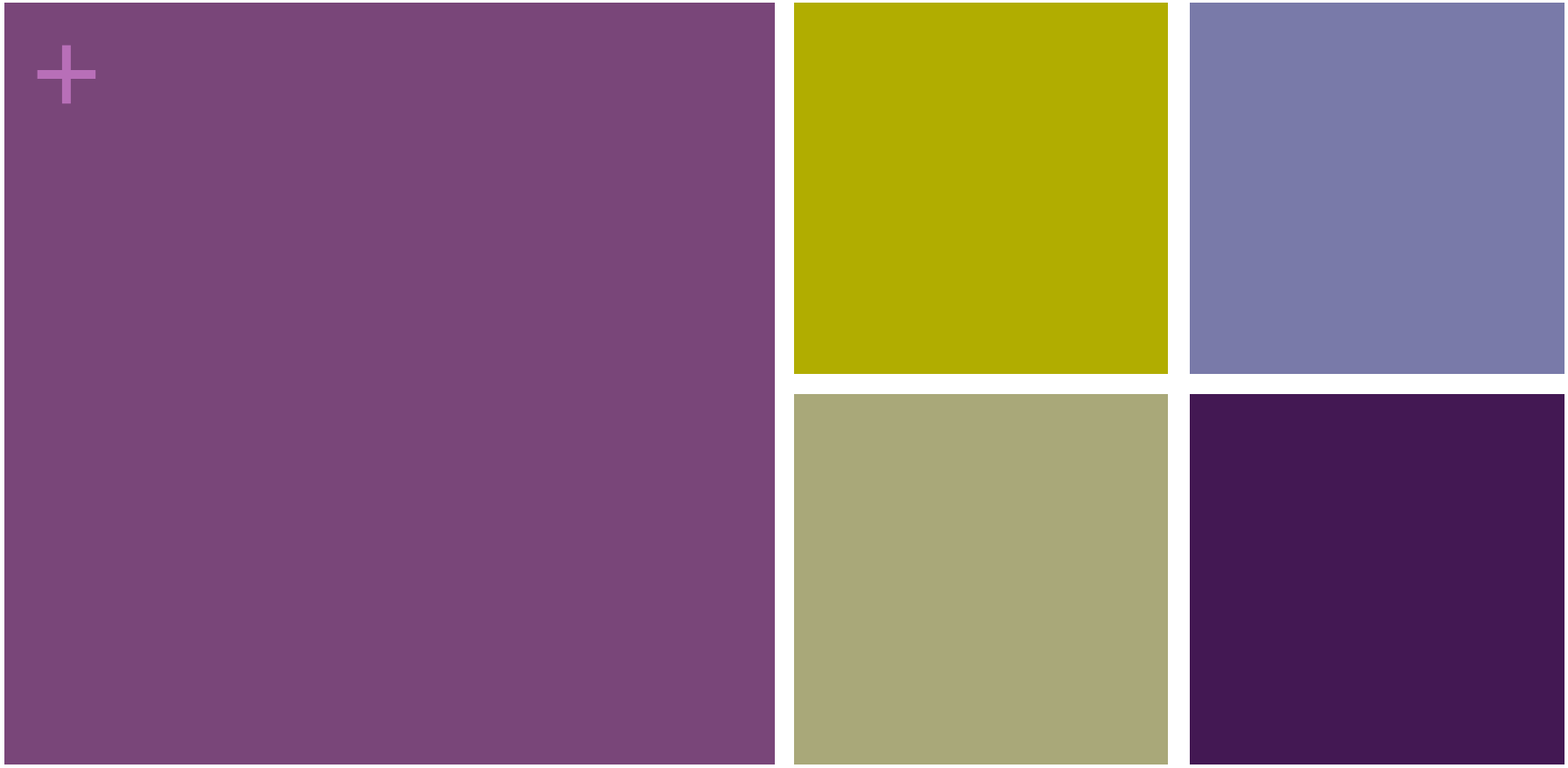


# Midterm - Review

Introduction to Computer Science CSCI UA.0101

Lecture 11

# Material (chapters 1–8 in the textbook)

- Input/Output from Console and Dialog Boxes
- Primitive types: int, double, char, boolean
- Assignment statements and expressions (Math class)
- Boolean expressions (relational and logical operators)
- Conversion between different types
- Operator precedence
- String class: string methods and processing
- Selection structures: if, if/else, if/else if and switch
- Repetition structures: for, while, do/while
- Methods: definition, calling, argument passing, overloading
- 1D and 2D arrays (declaring, creating, processing)

# Midterm format

- True/False and multiple choice  questions
- Code reading (read code to determine its function)
- Code writing (you will write code)
- A cheat sheet will be allowed in the exam: one 8.5x11 two-sided page of notes.

# The Assignment Operator

- In Java, the assignment operator is the single equals sign (the same as in Python)
- You can assign a value to a variable by using the following syntax:

```
int myInteger = 100;
```

- You do not necessarily need to assign data to a variable when you declare it. For example, the following statement is valid:

```
int myInteger;
myInteger = 100;
```

# Numeric Operations

| Name | Meaning | Example | Result |
|---|---|---|---|
| + | Addition | 10 + 2 | 12 |
| - | Subtraction | 100.0 - 0.5 | 99.5 |
| * | Multiplication | 100 * 2 | 200 |
| / | Division | 1.0 / 2.0 | 0.5 |
| % | Remainder (mod) | 20 % 3 | 2 |

# The Scanner Class

- The Scanner class is not a "static" class like the System class. This means that you can't just use it. You first need to make an "instance" of it in your program. Here's how to get started:

```
import java.util.Scanner;

public class Welcome {
      public static void main (String args[])  {
            Scanner input = new  Scanner(System.in);
      }
}
```

- The "new" keyword tells Java that we want to make a new Scanner object. We usually refer to this as an "instance" or as an "object"

# Reading Numeric Data

| Scanner Method | Resulting Data Type |
|---|---|
| `nextByte()` | Reads a byte integer |
| `nextShort()` | Reads a short integer |
| `nextInt()` | Reads an integer |
| `nextLong()` | Reads a long integer |
| `nextFloat()` | Reads a float number |
| `nextDouble()` | Reads a double number |

# Reading String Data

- You can use the nextLine() method to read in a String from the console with a Scanner object
- The nextLine() method will continue to read data until it encounters a line break (i.e. the user hit the Enter key)
- Example:

```
Scanner input = new Scanner(System.in);

System.out.print("What's your name? ");
String name = input.nextLine();

System.out.println("Welcome, " + name);
```

# Numeric Type Conversions (casting)

- Java will always convert data from data types that support smaller ranges into ones that support larger ranges, if necessary.
- Java won't, however, go the other way. Consider the following:

```
System.out.println(1.0 / 2);   // 0.5
```

- In this case Java turned the integer 2 into 2.0 and performed a division operation. What if we wanted to treat 1.0 as an integer and perform integer division instead? We can use a technique called "casting" to tell Java to force a value to behave in a certain way. For example:

```
System.out.println( (int) 1.0 / 2);   // 0
```

# Numeric Type Conversions (casting)

- When you cast a variable from a smaller type into a larger type (i.e. int to double) you are "widening" the type. Java will do this for you automatically as necessary.

- When you cast a variable from a larger type into a smaller type (i.e. double to int) you are "narrowing" the type. You must do this explicitly using casting syntax as follows:

```
System.out.println( (int) 10.75 * 2); //  20
```

- When you narrow a type you lose some information in the process. For example, in the statement above the double value 10.75 will lose its fractional value and will be truncated to a whole number.

# Using a Message Dialog Box

- We can then use the JOptionPane to display text by calling the "showMessageDialog" method inside the JOptionPane class.

- Methods on other classes can be called using "dot" syntax. To call the "showMessageDialog" method on the JOptionPanel class you can do the following:

```
JOptionPane.showMessageDialog(null, "I love ");

String answer = JOptionPane.showInputDialog("Enter
Score");

int ans = JOptionPane.showConfirmDialog("Are you ready
for the Midterm");
if (ans == JOptionPane.YES_OPTION) {
JOptionPane.showMessageDialog(null, "Great!");}
```

- Must import JOptionPane from javax.swing;

# Converting Strings to Numbers and Numbers to Strings

- The JOptionPane.showInputDialog() method will always return a String (much like how the input() function in Python always returns a String)

- If you wish to use the inputted data in a numeric expression you first need to convert it into the appropriate data type.

```
// define a String
String numStr = "5";

// create an int version of the String
int numInt = Integer.parseInt(num);

// create a double version of the String
double numDouble = Double.parseDouble(num);
```

- To Convert back to string:

```
numStr = Integer.toString(numInt); //OR

numStr = String.valueOf(numInt);
```

# Special Escape Characters

| Escape Character | |
|---|---|
| \b | Backspace |
| \t | Tab |
| \n | Linefeed |
| \r | Carriage Return |
| \\ | Backslash |
| \” | Quote |

# Working with Time values

- All computers have a standard time keeping mechanism based on the "UNIX Epoch"
- The "UNIX Epoch" is 00:00:00 on January 1st, 1970 GMT. All computers count forward from this time, and it serves as a common reference point for time calculations.
- You can access the current GMT time via Java using the following method call:

```
long currentTime = System.currentTimeMillis();
```

- This method call will return the number of milliseconds that have elapsed since the UNIX Epoch.

# “if” statements

- An “if” statement can be constructed in Java using the following syntax:

```
if (boolean-expression)
{
      statement(s);
}
```

- Parenthesis are required around your Boolean expressions
- Unlike in Python, indentation does not matter, though you should follow standard Java formatting rules and style.
- An “if” statement does not require a semicolon.
- Having a block of curly braces is required for “if” statements that block contains more than 1 statement. It’s usually a good idea to always put curly braces in anyway.

# Two-way "if-else" statements

- A one-way "if" statement only allows you to respond to a condition if the Boolean expression in question evaluates to true

- If you want to respond to when the condition evaluates to false you can use a two-way "if-else" statement using the following syntax. Note that in the example below one of the two sets of statements is guaranteed to execute – if the original boolean expression evaluates to false then the else clause automatically is invoked.

```
if (boolean_expression)
{
   // then statements: boolean expression is true
}
else
{
   // else statements: Boolean expression is false
}
```

# Nested "if" statements

- "if" statements can be placed inside of one another to form "nested" if statements using the following syntax:

```
if (boolean_condition)
{
     statement(s);

     if (boolean_condition_2)
     {
          statement(s);
     }
     else
     {
          statement(s);
     }
}
```

# Re-writing nested "if" statements using "else if"

- Sometimes it's easier to work with nested "if" statements if they are re-written using "else if" syntax.
- Each boolean expression is checked in succession. If one is found to be true we execute its associated block and do not test the additional expressions.
- If no expressions evaluate to true we can execute the "else" block at the end. Note that the presence of an "else" block is not required, and if it was omitted we could potentially never execute any statement in the nested "if" structure.

```
if (boolean_expression)
{
        statement(s);
}

else if (boolean_expression)
{
        statement(s);
}

else
{
        statement(s);
}
```

# Relational Operators

| Relational Operator | Description |
|---|---|
| < | Less than |
| <= | Less than or equal to |
| > | Greater than |
| >= | Greater than or equal to |
| == | Equal to |
| != | Not equal to |

# Logical Operators

| Operator | |
|---|---|
| `&&` | “and” – logical conjunction |
| `||` | “or” – logical disjunction |
| `!` | “not” – logical negation |

# Operator Precedence

- Earlier we learned that Java supports the standard order of operations for arithmetic operations.
- The full operator precedence chart is as follows (items we have discussed already are highlighted)

| Operator | Description |
|---|---|
| **( )** | **Parenthesis** |
| (type) | Casting |
| ! | Not |
| ***, /, %** | **Mult, Div, Mod** |
| **+, -** | **Add, Sub** |
| <, <=, >, >= | Comparison |
| ==, != | Equality |
| ^ | XOR |
| && | AND |
| \|\| | OR |
| = | Assignment |

# The “switch” statement

- Java provides a “switch” statement to simplify these kinds of tasks. For example:

```
int status = 2;

switch (status)
{
      case 1: System.out.println(“Status == 1”); break;
      case 2: System.out.println(“Status == 2”); break;
      default: System.out.println(“Invalid!”); break;
}
```

# Ternary (Conditional) Operator (?:)

```
variable = (bool expr.) ? Value if true : value if false


Examples:

int a, b;
a = 10;
b = (a == 1) ? 20 : 30;
System.out.println(b);          //output: 30

b = (a == 10) ? 20 : 30;        //output: 20
```

# Decision Based Questions

```
int x = 4,  y = 5, z = 6;

if  (x<y  && z>x)
{
        z = 4;
}

else if  (x==y ||  z==6)
{
        z = 5;
}

else
{
        z = 6;
}
```

1. What is the final value of z?
2. Rewrite without else if
3. Write a small block of code similar to assignments (how would you modify x and y so we get z = 6)
4. Modify a small block of code to change result

# Math Class: trigonometric methods

| METHOD | Description | Example |
|---|---|---|
| Math.sin() | returns sine of an angle in radians | Math.sin(Math.PI/2)<br>returns 0.5 |
| Math.cos() | returns cosine of an angle in radians | Math.cos (Math.PI/3)<br>returns 0.5 |
| Math.tan() | returns tangent of an angle in radians | Math.tan (Math.PI/4)<br>returns 1 |
| Math.toRadians() | returns the value in radians of an angle in degrees | Math.toRadians(30)<br>returns 0.5236 (= PI / 6) |
| Math.toDegrees() | returns the value in degrees of an angle in radians | Math.toDegrees(0.5236)<br>returns 30 |
| Math.asin() | returns the angle in radians that has a given sine | Math.asin(0.5)<br>returns 0.5236 (= PI / 6) |
| Math.acos() | returns the angle in radians that has a given cosine | Math.acos(0.5)<br>returns 1.0472 (= PI /3) |
| Math.atan() | returns the angle in radians that has a given tangent | Math.acos(0.5)<br>returns 1.0472 (= PI / 3) |

# Math Class: exponent methods

| METHOD | Description | Example |
|---|---|---|
| Math.exp(*x*) | returns *e* raised to power of *x* | Math.exp(1) returns 2.71828 |
| Math.log(*x*) | returns natural log of *x* | Math.log(Math.E) returns 1 |
| Math.log10(*x*) | returns the log in base 10 of *x* | Math.log10 (10) returns 1 |
| Math.pow(*a*, *b*) | returns *a* raised to the power of *b* | Math.pow(2, 3) returns 8 |
| Math.sqrt(*x*) | returns the square root of *x* | Math.sqrt(4) returns 2 |

# Math Class: rounding methods

| METHOD | Description | Example |
|---|---|---|
| Math.ceil(*x*) | returns *x* rounded up to its nearest integer as a double | Math.ceil(2.1) returns 3.0 |
| Math.floor(*x*) | returns *x* rounded down to its nearest integer as a double | Math.floor(2.1) returns 2.0 |
| Math.rint(*x*) | returns *x* rounded to its nearest integer as a double | Math.rint(2.5) returns 2.0 (if equally close to two integers, it returns the even one) |
| Math.round(*x*) | returns Math.floor(*x* + 0.5) as an integer if *x* is a float or as a long integer if *x* is a double. | Math.round(2.5) returns 3 |

# Math Class: other methods

| METHOD | Description | Example |
|---|---|---|
| Math.min(*a*, *b*) | returns the minimum of the two numbers | Math.min(2.5, 3) returns 2.5 |
| Math.max(*a*, *b*) | returns the maximum of the two numbers | Math.max(2.5, 4.6) returns 4.6 |
| Math.abs(*x*) | returns the absolute value of *x* | Math.abs(-2.1) returns 2.1 |
| Math.random() | returns a random positive double >=0 and < 1. It does not take any arguments | Math.random() |

# Using Math.random()

- Math.random() generates a random double >= 0.0 and < 1.0
- Example 1:

```
(int) (Math.random() * 10)          //returns a radom int between 0 and 9
```

- Example 2:

```
50 + (int) (Math.random() * 50)  //returns a radom int between 50 and 99
```

# Using the Random class

- The Random class can also be used to generate a random number

- Before using the Random class we need to import it:

  **import java.util.Random;**

- We also need to create an object of the Random class:

  **Random randomNum = new Random();** //randomNum is a generator

- Methods of the Random class:

  **randomNum.nextDouble();** //generates random double >= 0.0 and < 1.0

  **randomNum.nextInt(n);** //generates random int >= 0 and < n

# Generating Random Ranges

- You can also use the Math class to generate random numbers in customized ranges(i.e. generate#'s between 5 and 25 instead of 0.0 and 1.0)

- To do this you can apply the following algorithm:
  - Identify the Max and Min values of the range
  - Construct a statement like the following:

```
Random r = new Random();

int num = r.nextInt(4) + 3

(num will be from 3,4,5 or 6)
```

# Random Class

- Random Class generates a number from 0 to the number specified minus one

  For example: //game you win when random number from 5 to 19 is less than 10

```
Random r = new Random(232);

int randnum;
randnum = r.nextInt(15) + 5;
if (randnum < 10)
{
  System.out.println("You win!");
}
else
{
  System.out.println("You lost!");
}
```

- Know what a seed is (the parameter of Random) and why it is useful (to repeat always the same random numbers)

# String Comparisons

- Java supports string comparison through a compareTo() and equals() methods

```
String str1 = "ABB";
String str2 = "ABC";
String str3 = "ABC";

if (s1.compareTo(s2)== 0)   //  does the contents of s1 = s2?
if (s1.compareTo(s3) > 0)   //  is s1 > s3  alphabetically  ?
```

- You cannot use relational operators to compare strings:

```
String str1 = kb.nextLine();

if (str1 == "Yes") {…}                  //this is wrong

if (str1.equals("Yes")){…}              //correct
```

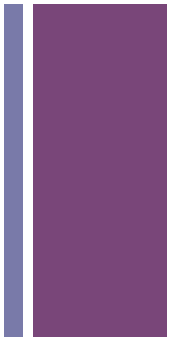
# String Class: methods for string processing

| METHOD | Description |
|---|---|
| str.length() | returns the number of characters in the string str |
| str.charAt(index) | returns the character at the specified index in the string str (the index of the first character is 0) |
| str.concat(s1) | returns a new string that concatenates str with s1 |
| str.toUpperCase() | returns a new string with all characters of string str converted to upper case. |
| str.toLowerCase() | returns a new string with all characters of string str converted to lower case. |
| str.trim() | returns a new string eliminating whitespace characters at both ends of the string str. |
| str.indexOf(s1) | returns the index of the first occurrence in string str of substring s1 (-1 if there is no such occurrence) |
| str.substring(index1, index2) | Returns a substring of string str starting at index1 and ending at index2. |

# Types of Loops

- Java supports 3 different loop structures
  - The "while" loop
  - The "do-while" loop
  - The "for" loop
- The 3 loops are interchangeable, and each has its own advantages that we will explore in today's lecture.

# “while” Loop Syntax



```
while (condition)
{
  // statement(s)
}
```

# The “do-while” loop

- The “do-while” loop is almost the same as a “while” loop except that it will execute its block before it evaluates its condition. For example:

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

- “do-while” loops are useful when you know that you want your loop to execute at least one time.
- Iterations: the number of times the loop is executed.

# The “for” loop

- Syntax:

```
for (int counter = 0; counter < 10; counter++)
{
     // statement(s)
}
```

- Note that in the above example we declared and defined ‘counter’ inside the loop. We could have just as easily defined ‘counter’ outside the loop, like this (just don’t do both!):

```
int counter = 0;
for (counter = 0; counter < 10; counter++)
{
     // statement(s)
}
```

# Infinite Loops

- When working with a "while" loop there is nothing to prevent you from writing a Boolean condition that will never evaluate to false. We call this an "infinite loop" since it never stops executing.
- If this happens your loop will continue executing forever, or until you send an "interrupt" to Eclipse using the "stop" button in the console panel.
- With the exception of a few special cases you want to try and avoid writing infinite loops

# Accumulator Variables

- Many programming tasks require you to calculate the total of a series of numbers or the number of times you iterate through a loop. We can accomplish this by creating an accumulator variable that can be updated over the lifetime of the loop.
- Initialize your accumulator variables outside of your loops.
- Decide on a value you want to initialize your accumulator values (at 0 works well if you are counting something)
- Use a self-referential assignment statement or augmented assignment statement when incrementing an accumulator variable. Example:
  - counter = counter + 1
  - counter += 1
  - counter++

# Loop Design Strategies

- Identify statements that need to be executed
- Wrap the statements inside a loop structure

```
while (loop continuation condition)
{
	// statements
}
```

- Code the loop continuation condition and add in appropriate statements for controlling the loop. You will most likely also need to set up control structures outside the loop.

```
// set up control structure

while (loop continuation condition)
{
	// statements

	// statements to control the loop
}
```

# The "break" Command

- The break command can be used to immediately end a loop
- It's not required to ever use break – you can always simulate its function by adjusting your loop control condition
- Example:

```
int sum = 0;
while (true)
{
      sum += (int) (Math.random() * 11);
      System.out.println("sum = " + sum);
      if (sum % 13 == 0)
      {
            System.out.println("Found a multiple of 13!");
            break;
      }
      System.out.println("Looping ....");
}
```

- How would you change the code so we don't have to use a break?

# The "continue" Command

- The "continue" command can be used to cause a loop to end its current iteration, but not end the loop. For example:

```
// sum up all numbers between 1 and 100, except multiples of 10

int counter = 0;
int sum = 0;
while (counter < 100)
{
        // increase counter
        counter++;

        // is this a multiple of 100?
        if (counter % 10 == 0)
        {
                // skip it
                continue;
        }
        else
        {
                sum += counter;
        }
}

System.out.println("Sum of 1 - 99: " + sum);
```

- How would you change the code so we don't have to use a continue?

# Nested Loops

- A nested loop is a "loop inside another loops"
- We usually refer to nested loops in terms of their "outer" and "inner" loops. The outer loop is the first loop that is encountered in your program, and the inner loop is the loop that is nested inside of the outer loop. For example:

```
for (int outer = 0; outer < 10; outer++)
{
      // statement(s)

      for (int inner = 0; inner < 10; inner++)
      {
            // statement(s)
      }

      // statement(s)
}
```

# Loop Questions

```
int x = 4, y = 5, z = 8;

for (int outer = x; outer < y; outer++)
{
      z = z + 1;

      for (int inner = 0; inner < 5; inner++)
      {
            z = z + 1;
      }
}
```

1. What is the final value of z?
2. How many iterations are performed on the inner loop?
3. Write the above as a while loop

# Defining a Method

- Refer to the following method header syntax:

```
modifier returnTypeValue methodName (parameters) {
        // statement(s)
}
```

- The **methodName** for a method is a name of your choosing. Be sure to conform to the Java standards for identifiers (no spaces, no leading numbers, etc)

- The **parameters** for a method are placed within the parenthesis after the methodName. These are values that you will be passing into the method when you invoke or call it. Methods can accept zero or more parameters. For example:

```
// no parameters
public static void method1 () { }

// 1 parameter
public static void method2 (int a) { }

// 2 parameters
public static void method3 (int a, int b) { }
```

# Calling a Method

- A method definition is used to define what you want a method to do.
- In order to run the statements inside your method you must invoke or "call" the method from another part of your program.
- If your method does not return a value you can simply type its name along with any parameters as a statement. For example:

```
// call the awesome method which does not have
// a return value
awesome();
```

- If your method does return a value you can treat the method as though it was that value. For example:

```
// call our method and store the result
int answer1 = addTwoNumbers(5, 7);

// call our method and use the result immediately
int answer2 = 100 + addTwoNumbers(50, 51);
```

# Passing Parameters by Value

- The power of methods lie in their ability to accept parameters – or arguments – from a caller and then operate on this data.
- When designing methods you need to articulate the type of data you will be sending that method along with the order in which it can be expected. This is commonly referred to as "parameter-order association"

# Variable Scope

- A variable's "scope" is the region of the program where the variable can be referenced by your code.
- In Java, variables defined inside of a method are considered "local" to that method and are inaccessible outside of the method.
- In addition, variables declared inside a block are local to that block and cannot be referenced outside of that block (but the can be referenced by blocks nested inside of the block)
- Variables defined in the initialization of a "for" loop are local to the entire loop, but are inaccessible outside the block of the loop.

# Overloading Methods

- Java supports a technique called "overloading" that lets you define multiple versions of the same method that can be invoked using different parameter sets.

- When overloading a method you simply define a different method for each version you wish to create, making sure that each version uses the same name but different parameter sets. Java will figure out at compile time which method to invoke based on the data provided. Example:

```
public static int addTwo (int a, int b)

public static double addTwo (double a, double b)
```

# Sample Method Problems

- Given the following code

```
public static void main (String [] args){
    int x = 4, y = 5, z = 8;
    x = mymethod(y, z, 8) + y

}
public static int mymethod(int ina, int inb, int inc){

    ina = inb * 3;
    inb = ina - 1;
    inc = ina + inb;
    return inc;
}
```

1. What is final values of x, y, z?
2. Modify code to perform another task.

# Declaring an Array

- Arrays, like all variables, must be declared prior to use.
- You can declare an Array using the following syntax. Note the bracket notation after the data type:

```
elementType[] arrayReferenceVariable;
```

- "elementType" can be any data type you want. The trailing brackets after the data type tell Java that this variable will be treated as a collection of items of those data types.
- All elements in an array must be of the same type (i.e. you can't have an array that holds both ints and doubles)
- For example, if you want to declare an Array of integers you could do the following:

```
int[] myIntegerArray;
```

# Creating Arrays

- Declaring an array is not enough to begin using it in your program. Before you use an array you need to tell Java to create space in memory for all of the elements it will be holding.

- You can do this using the following syntax:

```
elementType[] arrayReferenceVariable;
arrayReferenceVariable = new elementType[size];
```

- The "size" value determines the number of items that your array can hold. Note that arrays cannot grow beyond this size, unlike in Python. We will discuss techniques to get around this later in this chapter.

# Processing Arrays with "for" loops

- We almost always use a 'for' loop when working with our arrays. 'for' loops give you the ability to easily iterate over a known range, and we can construct one that will visit every element in an array by using the 'length' property of the array in the iteration condition. Here's an example:

```
int myList = new int[5];

for (int c = 0; c < myList.length; c++)
{
     // print out what is at element c
     System.out.println( myList[c] );
}
```

- You should know how to populate an array with values entered by the user and how to display those numbers in the console.

- For numerical arrays you should know how to calculate totals, average, as well as the largest and smallest number in the array.

# The Enhanced For Loop

- The "enhanced for loop" allows you to iterate through an array or collection without having to create an iterator or without having to calculate beginning and end conditions for a counter.

```
double [] numbers = {1, 2, 3, 4, 5};

for (double n: numbers)
{
	System.out.println(n);
}

//is equivalent to

for(int i = 0; i < numbers.length; i++)
{
	System.out.println(numbers[i]);
}
```

# Array Length

- Arrays know their own length (i.e. number of elements).
- When an array is created it is created with a certain number of elements. Arrays cannot grow beyond this size.
- You can access the size of any array by using the array's "length" property. For example:

```
myList = new int[5];
System.out.println( myList.length );

// will print 5
```

# Passing Arrays to Methods

- Arrays can be passed to methods in the same way that primitive data types can be passed to methods. For example, the following method accepts an array as an argument:

```
public static void doSomething(int[] myList)
```

- `myList` is a reference variable, not a primitive data type. This means that `myList` contains the memory location of the array not the values of the elements of the array.

# Arrays as a reference type

- Arrays are considered a reference type in Java (i.e. arrays are objects in Java).
- This means that the array variable doesn't contain the actual data associated with the array. It simply stores the memory address of where the array can be found.
- When Java passes an array into a method it is passing this reference to the array, not the array itself. Because Java passes by value, it is passing the value of the array reference (the memory location) which results in a situation where a method can change the contents of the array without having to return it to the caller. Your book refers to this behavior as "passing by sharing"

# Difference between primitive types and reference types

- Primitive types store the actual values (data) of the variables. Examples of primitive data types:

```
int a;
float b;
double c;
char d;
boolean e;
```

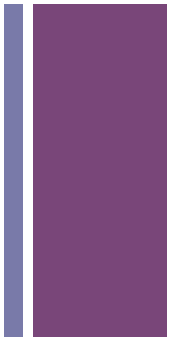
**a**, **b**, **c**, **d**, and **e** are primitive types

- Reference types store the memory address where the objects they refer to are located in the heap memory; not the object data itself. Examples of reference types:

```
Scanner kb;
String words;
int [] numbers;
```

**kb**, **words** and **numbers** are reference types.

# Variable Length Argument Lists



- You may only have one variable length argument in a method header, and it must always be the last argument. For example:

  **public static void method1(int a, double …b)  //OK**

  **public static void method2(String …a, int b)  //error (vararg should be last)**

  **public static void method3(String…a, double …b)  //error (two varargs)**

- If you omit the variable length argument list when calling a method Java will simply create an empty array and send it to the method – no exception will be raised.

# Copy Method 1: Looping

- One way to copy an array is to use a loop to copy array elements individually. For example:

```
int[] a = {1,2,3};
int[] b = new int[3];

for (int x = 0; x < a.length; x++)
{
     b[x] = a[x];
}
```

# Copy Method 2: System.arraycopy

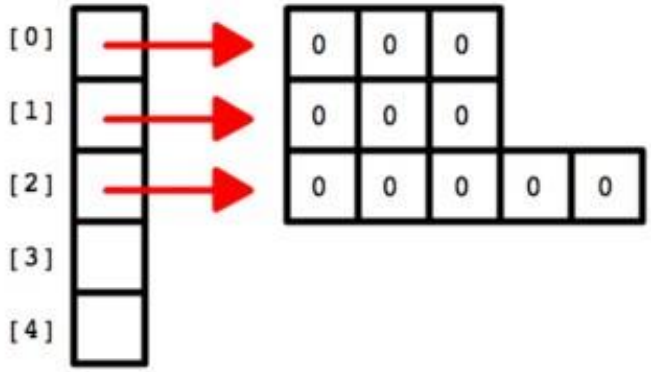
- The System class in Java contains a method called "arraycopy" that allows you to copy over information from one array to another using.
- The syntax for using this method is as follows:

```
System.arraycopy(sourceArray, sourcePosition,
                 destArray, destPosition,
                 length)
```

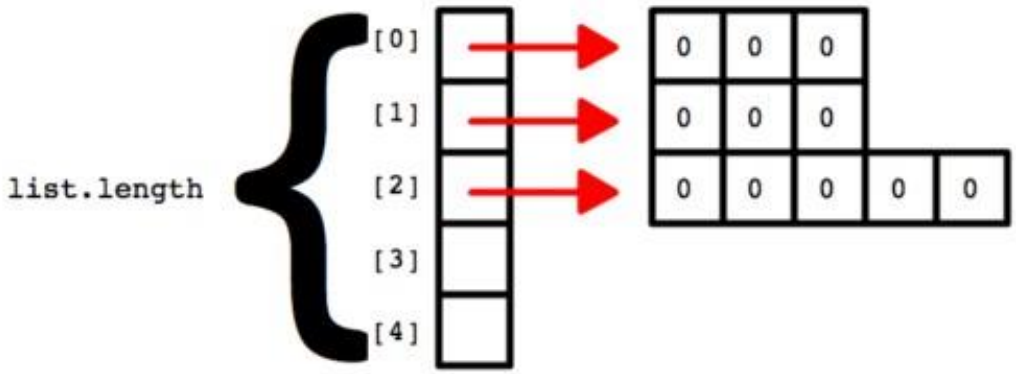
- Where sourceArray is the array you wish to copy from, sourcePosition is the postion you want to start copying from, destArray is the array you wish to copy to, destPosition is the position you want to start placing elements, length is the number of elements you wish to copy.
- Note that you must first create destArray if you want to use System.arraycopy – the method won't allocate space on the heap for your new array.

# Getting the length of a two dimensional array

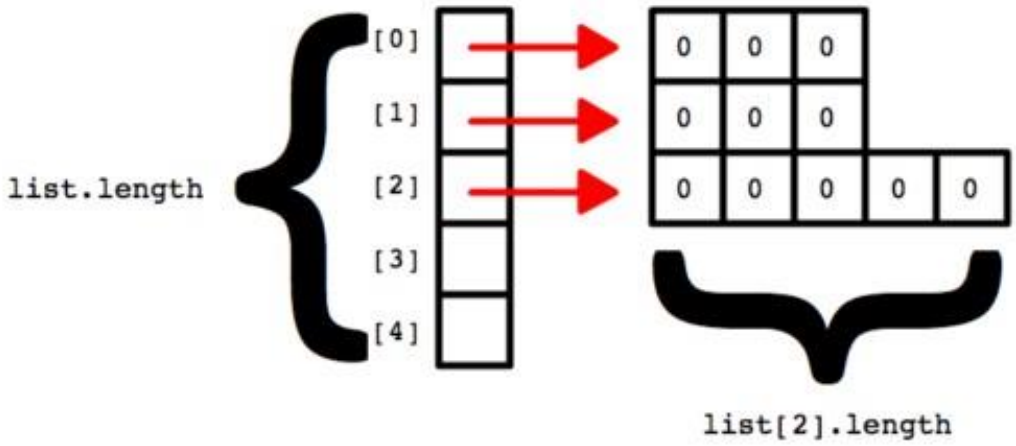
```
int[][] list = new int[5][];

list[0] = new int[3];

list[1] = new int[3];

list[2] = new int[5];
```



# Getting the length of a two dimensional array



# Getting the length of a two dimensional array



# Let’s work on the Practice Test …