

---

# Chapter 5

## Methods & Classes

### Part 1: Static Methods

---

Adapted from notes by Pat Baker based on Java Foundations by Lewis, Chase, & DePasquale

# Chapter Topics

- Working with methods
- Identifying classes and objects
- Structure and content of classes
- Instance data
- Visibility modifiers
- Constructors
- Relationships among classes
- Static methods and data

# Modular Programming

- A **method** is a collection of one or more statements that performs a specific task
- As programs become more complex, it is useful to break the program into smaller, more manageable methods that do part of the work
  - This is especially true for repetitive tasks that appear in the code
- We call this **modular programming** and it has many benefits:
  - Writing programs becomes simpler
  - There is less redundant code
  - Programs become more readable
  - It is easier to maintain the code

# Defining and Calling Methods

- **Method definition/declaration:** the statements that make up a method
  - Note that in some languages such as C++ there is a distinction between the method declaration & its definition
  -
- **Method call:** a statement that causes a method to execute

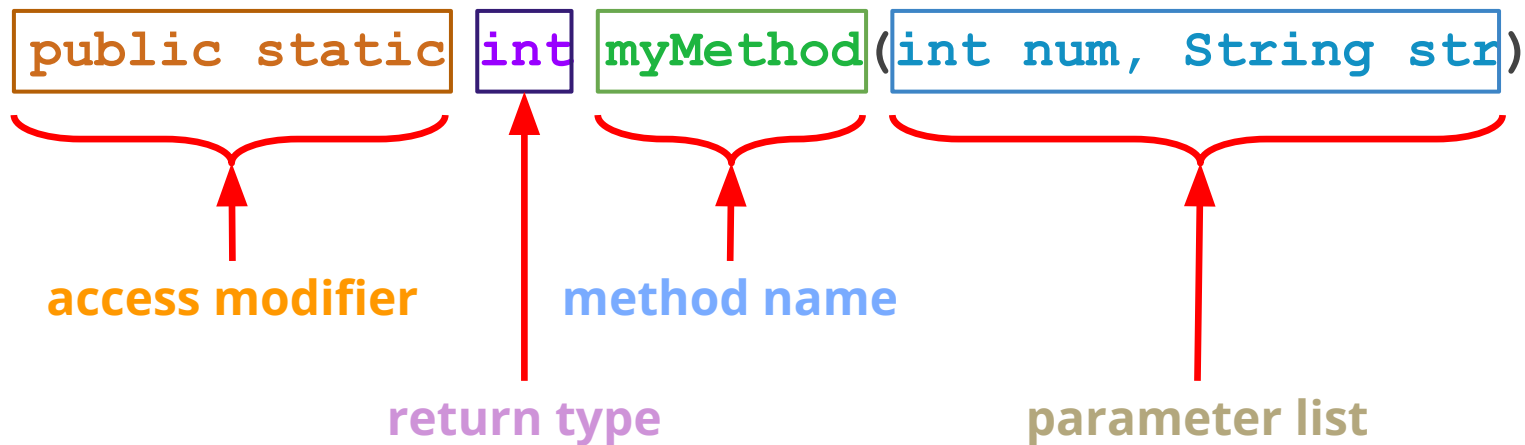
# Method Definitions

- The method definition (declaration) specifies the code when the method is **invoked** (called)
- The general structure of a method definition is as follows:

```
access_modifier return_type method_name(formal_parameters) {  
    // Method body (code block)  
}
```

# Method Definitions

The method definition begins with a **method header**



The method should return a value that is the same as the return type  
For example, `return num * 4;`

# Method Definition - Example

The following is commonly written code to compute the mean of an array:

```
int sum = 0;
double average;
for (int ii=0;ii<numbers.length;ii++) {
    sum += numbers[ii];
}
average = (double) sum / numbers.length;
System.out.println("Average = " + average);
```

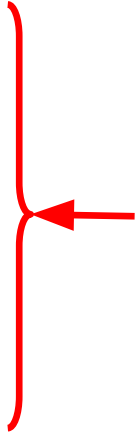
But suppose you had several arrays that you need to compute the average for - that would take a *separate* loop for each of the arrays!

A better way would be to write a method

# Method Definition - Example

The following code defines a method to print the average of an array:

```
public static void printMean(int[] values) {  
    int sum = 0;  
    double average;  
    for (int ii=0;ii<numbers.length;ii++) {  
        sum += numbers[ii];  
    }  
    average = (double) sum / numbers.length;  
    System.out.println("Average = " + average);  
}
```



method  
body



# Method Definition - Example

The following code defines a method to print the average of an array:

```
public static void printMean(int[] values)
{
    int sum = 0;
    double average;
    for (int ii=0;ii<numbers.length;ii++)
        sum += numbers[ii];
}
average = (double) sum / numbers.length;
System.out.println("Average = " + average);
}
```

**But, notice we print the average. Why not let the *caller* decide what to do what they want with the result?**

# Method Definition - Example

The following code *returns* the average of an array (notice the return type changes from `void` to `double`):

```
public static double mean(int[] values) {  
    int sum = 0;  
    double average;  
    for (int ii=0;ii<numbers.length;ii++) {  
        sum += numbers[ii];  
    }  
    return (double) sum / numbers.length;  
}
```

# return statement Summary

- The return type of a method indicates the type of value that the method sends back to the caller
- A method that does not return a value has a `void` return type
- A return statement specifies the value that will be returned

`return expression;`

- Its expression must conform to the return type. The type in the return expression must be the same as the type in the method header

# Calling a Method

To call/invoke a method, use the method name followed by ( )

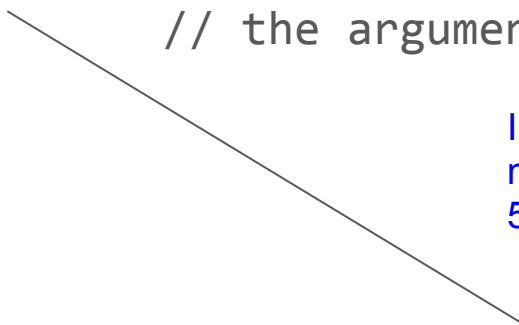
```
double result = milesToLaps( 52.3 );
```

When a method is called, the **actual arguments**/parameters in the invocation are copied into the **formal parameters** in the method header

# Calling a method

```
double result = milesToLaps( 52.3 ); // call the method - 52.3 is  
                                     // the argument (actual parameter)
```

In this example, when  
milesToLaps is called it places  
52.3 into userMiles



```
public static double milesToLaps(double userMiles) //method header  
                                                    //userMiles is the formal parameter
```

↑  
will return  
a double

# Recap of Local Data in methods

- As we've seen, local variables can be declared inside a method
- The formal parameters of a method become automatic local variables in the method
- When the method finishes, all local variables are destroyed (including the formal parameters)

# Overloading Methods

- **Overloaded methods** are two or more methods that have the *same name*, but *different parameter lists*
- The **method signature** include the number, type and order of the method parameters
- Overloaded methods are used to create methods that perform the same task, but take different parameter types or a different number of parameters
- If a method is overloaded, the method name by itself is not sufficient to determine which method is being called
- The compiler will determine which version of the method to call based on the argument and parameter lists
- The return type of a method is not part of the signature and has no bearing on overloading

# Overloading Methods

Methods are overloaded if:

1. They have the same method name
2. They have a different parameter list, so they must have
  - different number of parameters
  - if they have the same number of parameters, then the parameter data types must be distinguishable

Remember that the return datatype is not considered in whether methods are overloaded



# Method overloading

Defining overloaded methods:

```
public static int process(    ) //1  
{ //does something }
```

```
public static int process(int x) //2  
{ //does something else }
```

```
public static int process(int x, double y) //3  
{ //does something else, or the same thing but with more parameters }
```

Calling overloaded methods:

```
int value = process();           // calls the process method 1 that has no formal parameters  
int result = process(22);        // calls the process method 2 that has one int parameter  
int val = process(5, 367.23);    // calls the process method 3 that has one int parameter and one double parameter
```

# Method overloading

Defining overloaded methods:

```
public static int process(int x)    //1
{ //does something great  }

public static int process(String name) //2
{ //does something great  }
```

Calling overloaded methods:

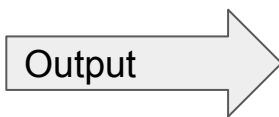
```
int result = process( 2);    // calls the process method 1 that has one int parameter
String x = process( "spring");    // calls the process method 2 that has one String
parameter
```

# Passing by Value

- Java always passes variables **by value**
- Variables are *always* primitive data types (**int, double, char, boolean...**)
- All of the methods we have considered so far have been examples of passing by value
- So what does this mean?
  - A *copy* of the passed-in variable is copied into the argument of the method
  - Any changes made inside the method do not affect the original variable.

# Passing by Value

```
public static void increment(int value){  
    value = value + 10;  
    System.out.println("Inside the method, value is: " + value);  
}  
  
public static void main(String[] args) {  
    int x = 5;  
    System.out.println("Before calling the method, x is: " + x);  
    increment(x);  
    System.out.println("After calling the method, x is: " + x); }  
}
```



```
Before calling the method, x is: 5  
Inside the method, value is: 15  
After calling the method, x is: 5
```

So what is happening? In the main, x is 5. The call to the **increment** method copies what is in x (happens to be a 5) into the method parameter **value**. The **increment** method changes **value**. But when execution returns to the main, we see that x has not changed!

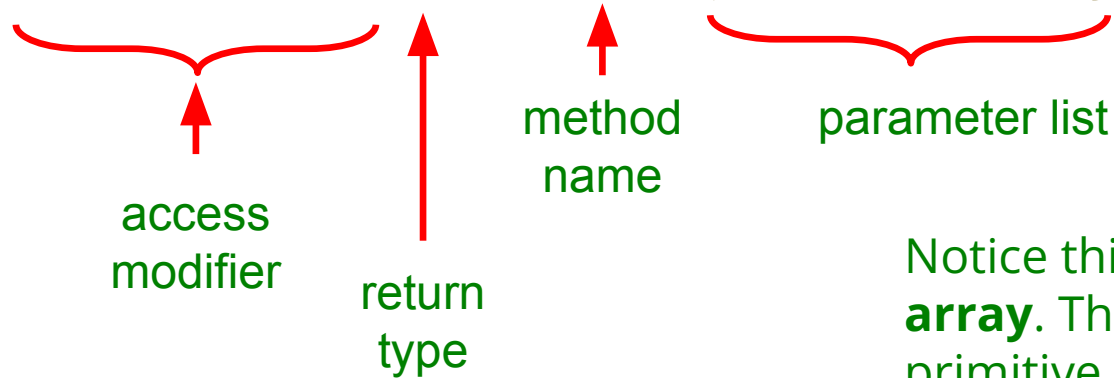
# Passing by Value vs Passing by Reference

- A method argument that is a primitive type (int, double, char, etc) is passed to the method parameters by pass-by-value: copying the value into the parameter
  - Any assignments or changes made to the parameters inside the method do not affect the original arguments
- But when you pass an array into a method, it passed to the method by **pass-by-reference** - copying *a reference* to the object through the parameter
  - The argument is an **alias** of the passed-in array
  - Any changes to the argument will affect the original array
- In general, if an object is passed into a method, it is passed **by reference**

# Method Header - Pass by Reference Example

- A **method definition** begins with a *method header*

```
public static int findMax (int[] someArray)
```




Notice this parameter is an **integer array**. The parameter is an object, not a primitive data type.

`someArray` is called a *formal parameter*

# Method Body - Pass by Reference Example

```
public static int findMax (int[] someArray)
```

```
{  
    int max = someArray[0]; //set the initial max  
    for(int i = 0; i < someArray.length; i++)  
    {    if (someArray[i] > max)  
        max = someArray[i];  
    }  
    return max;  
}
```



max and someArray are local data

They are created each time the method is called, and are destroyed when it finishes executing

The return expression  
must be consistent with  
the return type

# Calling a Method

Given the following:

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

- To call/invoke a method, use the method name followed by ( )

```
System.out.println("max :" + findMax(exams) );
```

or

```
int maximum = findMax(exams);
```

- When the method is called, the array object *reference* is passed to the *formal parameter* in the method header. In this example, the reference to the **exams** array is passed into method.



# Calling a Method - Pass by Reference Example

## How it works

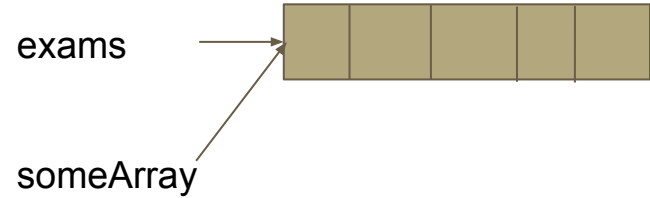
Given the following:

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

```
int maximum = findMax(exams); //call the method
```

```
public static int findMax (int[] someArray) //method header
```

In this example, both **exams** and **someArray** point to the same array. Their references are the same. Any changes made to **someArray** in the method affect **exams**.



# Pass by Reference Example

```
public static void increment(int[] someArray)
{
    for(int i = 0; i < someArray.length; i++) //add 10 to every element of someArray
        someArray[i] = someArray[i] + 10;

    System.out.print("\nInside the method, the score array is: ");

    for(int temp : someArray) //display contents of someArray
        System.out.print(temp + " | ");
}

public static void main(String[] args) {

    int[] score = {100, 78, 45}; //creates an array with 3 elements

    System.out.print("\nBefore calling the method, the score array is: ");
    for(int tempScore : score) //using for each loop to display contents of score
        System.out.print(tempScore + " | ");

    increment(score); //call the method and pass by reference

    System.out.print("\nAfter calling the method, the score array is: ");

    for(int tempScore : score) //display contents of score
        System.out.print(tempScore + " | ");
}
```

## Output:

Before calling the method, the score array is:100 | 78 | 45

Inside the method, the score array is: 110 | 88 | 55

After calling the method, the score array is: 110 | 88 | 55

# Using *method stubs* to assist with program development

- A **method stub** is a method definition whose implementation statements have not yet been written.
- Using this technique supports “divide-and-conquer”
- Method stubs compile and allow the developer to implement incrementally

Some method stub examples. These would all compile:

```
public static void
instructions(int cycles)
{
    //implementation goes here
}
```

```
public static void
swapValues(int[ ] values)
{
    //implementation goes here
}
```

```
public static int compute(int num1, int num2)
{
    //implementation goes here
    return 0;        //0 is just a placeholder
}
```

```
public static double kelvinToCelsius(double
valueKelvin)
{
    //implementation goes here
    return 0.0;    //0.0 is just a placeholder
}
```

# Method Design

- An **algorithm** is a step-by-step process for solving a problem
- Every method implements an algorithm
- Consider the **pre-** and **post- conditions**
  - Preconditions are the things that are true *before* the method is invoked
  - Postconditions are things that are true *after* the method is invoked
- These form the basis of **test cases**
  - You can write tests that set up certain preconditions then evaluate that the postconditions hold after the method is invoked

# Designing Methods

- Some best practices for designing methods
  - Use methods to avoid duplicating code
    - If you see duplicate code, think about how you could put that code in a method
  - Methods should usually be short and should perform one single task
    - There is essentially no cost for having more methods
  - Use descriptive names
  - The fewer arguments the better
  - Avoid **side-effects** (unrelated changes)
    - Your methods should not change things unless necessary
      - And it is usually not necessary!









# While Loops

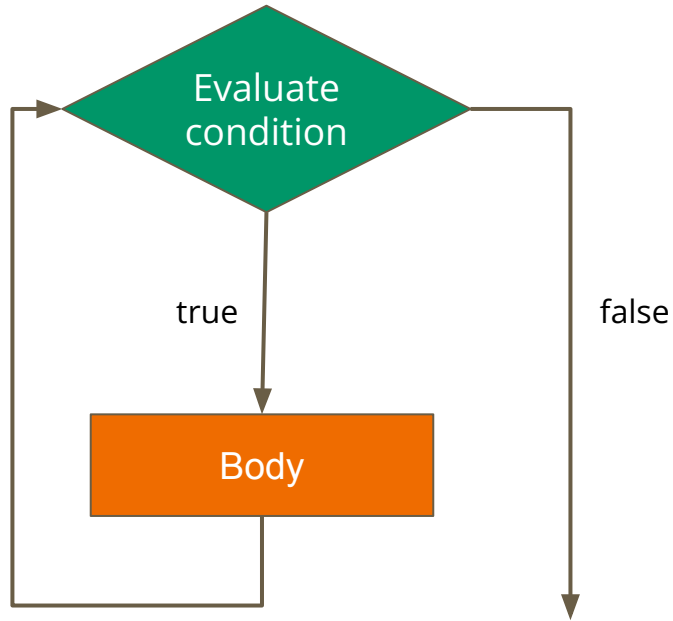
- The while loop has the following syntax:

```
while (condition) {  
    // loop body (code block)  
}
```

- While the condition is true, the code block in the **loop body** is repeated
- Each time the code block executes, the condition is reevaluated
- The loop continues until the condition becomes false
- It is essential that the condition has the possibility of becoming false as the loop body executes
- The body of a while loop will execute 0 or more times
  - If the condition is false, it will not execute at all

# While Loops

Logic flow of a while loop:



# While Loops

Example code:

```
int num = 1;
while (num <= 10) {
    System.out.println(num);
    num++;
}
```

This code prints the numbers from 1 to 10, inclusive

Note that if we set num to a number greater than 10, the loop would never execute!

# Applications of Loops

We can use a loop to calculate a running sum:

```
int num = 1;
int total = 0;
while (num <= 10) {
    total = total + num;
    num++;
}
System.out.println("The total is " + total);
```

Notice that we initialize total to 0, then add the current num to the running total during each loop iteration

# Applications of Loops

We can use a loop to get an arbitrary number of values:

```
int value = -1;
int total = 0;
System.out.print("Enter a positive number (0 to quit): ");
value = scnr.nextInt();
while (value != 0) {
    total += value;
    System.out.println("  The total so far is " + total);
    System.out.print("Enter a positive number (0 to quit): ");
    value = scnr.nextInt();
}
System.out.println("The total is " + total);
```

Here, we call `-1` a **sentinel value** because it is used to signal the end of the input

# Applications of Loops

We can use a while loop to validate input:

```
int num = -1;
while (num <= 0) {
    System.out.println("Enter a positive number: ");
    num = scan.nextInt();
}
System.out.println("The total is " + total);
```

The loop will print the prompt and scan a new integer until the user enters a positive number

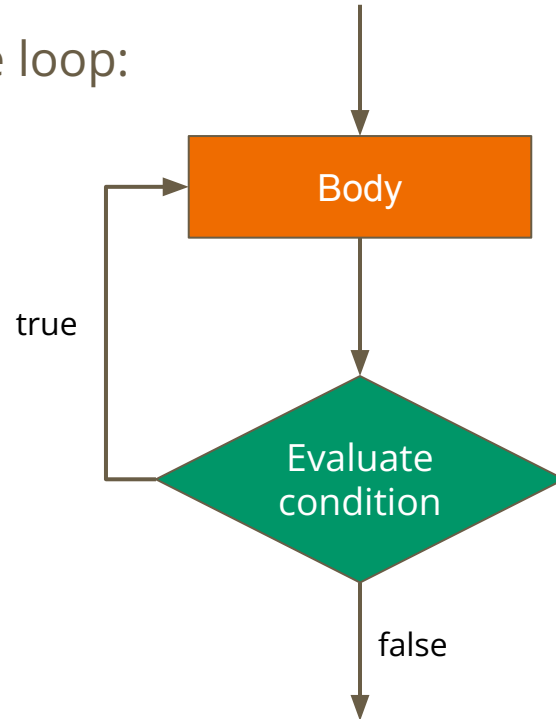
# Do-While Loops

- The do-while loop executes the loop body *before* checking the loop condition
- It has the following syntax:

```
do {  
    // loop body (code block)  
} while (condition);
```
- The code block in the loop body is executed once initially, and then the condition is evaluated
- The code block in the loop body is then executed repeatedly until the condition becomes false
- The body in a while loop will execute *at least* once
  - If the condition is false, it will only execute once

# Do-While Loops

Logic flow of a do-while loop:





# Do-While Loops

Example code:

```
int num =0;  
do {  
    num++;  
    System.out.println(num) ;  
} while (num < 10)
```

This code prints the numbers from 1 to 10, inclusive

Note that if we set num to a number greater than 10, the loop would never execute!

# For Loops

- The for loop has the following syntax:

```
for (initialization; condition; update) {  
    // loop body (code block)  
}
```

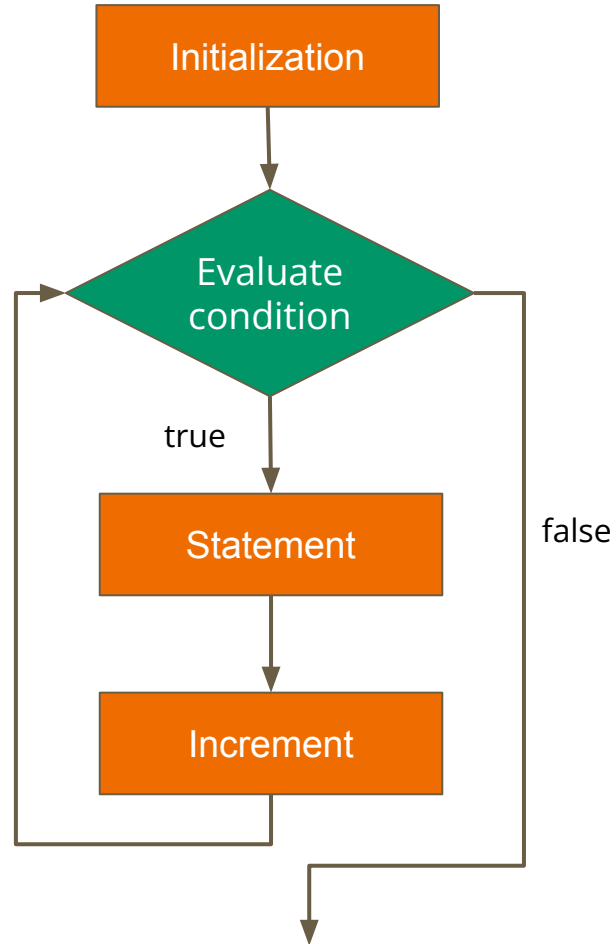
- The **initialization** code is executed *one time* when the loop is entered
- The code block in the body is executed until the **condition expression** becomes false
- The **update** code is executed at the *end* of *each* iteration
  - Typically the update code would increment a variable used in the condition expression
- Multiple expressions can be evaluated in each part if separated by a comma
- For loops are best suited for executing statements a predetermined number of times

# A Note About Loop Variable Names

- The name you give variables should reflect *what they represent*
- In some cases with loops, the variable just represents how many times we have iterated through the loop
  - Most commonly with for loops, but sometimes with while loops
- In those cases, programmers often choose simple names like `i`, `j`, & `k`
- My personal preference is to double the letter (so `ii`, `jj`, & `kk`) to make it easier to see & search for (`i` is a pretty common character, but `ii` rarely shows up so is easier to find/replace)
  - This is just *my* preference - decide for yourself how you want to name loop variables

# For Loops

Logic flow of a for loop:



# Example For Loop

- We can declare a variable in the initialization
- The update section can perform any calculation(s)

In this loop, we declare an int variable called num in the initialization and we increment num at each iteration

```
for (int num = 0; num <= 10; num++) {  
    System.out.println("num=" + num);  
}
```

# Example For Loop

This loop is similar to the previous loop, but here we update num by adding 5 each time

```
for (int num = 0; num <= 100; num+=5) {  
    System.out.println("num=" + num);  
}
```

# Example For Loop

We can use a loop to calculate a running sum (equivalent to the earlier while loop example):

```
int total = 0;
for (int num = 1; num <= 10; num++) {
    total = total + num;
}
System.out.println("The total is " + total);
```

# Example For Loop

- You can have multiple statements in each section of the loop

Here, we have 2 declarations in the initialization and 3 statements in the decrement:

1. The variable `ii` is **increased** by 7 each iteration
2. The variable `jj` is **decreased** by 13 each iteration
3. The variable `kk` (*from outer scope*) is doubled each iteration

```
for(int ii=0, jj=100;ii<jj;ii+=7, jj-=13, kk*=2){  
    System.out.println(" ii= " + ii + ", jj= " + jj + ", kk= " + kk) ;  
}
```



# Example For Loop

- Each expression in the for-loop header is optional
  - If you leave out the initialization, no initialization is performed
  - If you leave out the condition, then condition evaluates to true, creating an infinite loop
  - If you leave out the update, no update operation is performed, so the update operation needs to be performed inside the loop to avoid an infinite loop

```
for ( ; ; ) {  
    System.out.println("Infinite loop!");  
}
```

# For Loops vs. While Loops

For loops are functionally equivalent to while loops

So the following for loop

```
for (initialization; condition; update) {  
    // loop body  
}
```

is equivalent to the following while loop

```
initialization;  
while (condition;) {  
    // loop body  
    update;  
}
```

For loops are better suited in situations where the number of iterations is known

# While Loops vs For Loops

- Should you use while loops or for loops?
  - It depends
- In general, use a for loop if you know how many iterations you want to have *before* the loop starts
  - Add the numbers from 1 to N
  - Print every record in a collection
- Use a while loop when you do not know how many iterations you want to have until *after* the loop starts
  - Read each line of a file
  - Add all the numbers a user enters until they enter a negative number

# Infinite Loops

- The **body of a loop must eventually make the condition false**
- If it does not, this creates an **infinite loop** that executes until the user interrupts the program
- It is *very* easy to create an infinite loop, so double check the logic of your loops carefully
  - Always make sure that the condition will eventually become false to allow the loop to terminate
- Again, the **loop body must eventually make the condition false**

# Infinite Loops

Example infinite loop:

```
int num = 0;
while (num < 10) {
    System.out.println("Loop without end");
    num = num - 1;
}
```

Since we are subtracting 1 from num each iteration, this is an infinite loop

# Infinite Loops

Example infinite loop:

```
while (true) {  
    System.out.println("The loop that's on everybody's nerves");  
}
```

This loop will print “Loop without end” until the program is interrupted with a Ctrl-C

There *are* situations where you would want to do something like this, but those situations are rare, so if you have to ask then no, this isn't that situation

# Nested Loops

- Recall that one similarity between conditionals and loops is that they both contain a code block as their body
- Code blocks can contain any valid code, so we can have a loop in the body of another loop
  - We can have for loops in while loops, while loops in do-while loops, etc.
- We can also have if statements in loop bodies, loops in if statement bodies, etc.
- The inner loop iterates completely one time for **each** time the outer loop iterates

# Nested Loops

- A **nested loop** is a loop that appears in the body of a loop.
- The **outer loop** contains an **inner loop** in its body
- Nested loops are useful for generating all combinations of a set of items
  - For example, print a rectangle of \* characters 1 row at a time
  - Print a matrix where each element is 0
  - Compute the sum of every pair of numbers in a range
- Note that there is no limit to how deeply you can nest loops, though practically each nested loop decreases the performance of your program substantially, so only use them if you need to

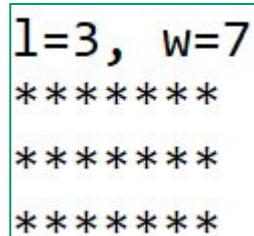


# Nested Loops

The nested loop below prints a  $l \times w$  rectangle of `*` characters with a width of `w` & a length of `l`

- The outer loop iterates over the number of rows (# rows = the length)
- The inner loop iterates over the number of columns (the width)
  - Its body prints one `*` during each iteration

```
for(int ii = 1; ii <= l; ii++) {  
    for(int jj = 1; jj <= w; jj++) {  
        System.out.print("*");  
    }  
    System.out.println( );  
}
```



l=3, w=7  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

The `println()` at the end of the outer loop's body starts a new row

# Nested Loops

The nested loop below calculate a times table

- The first (**outer loop**) iterates over the number of rows
  - Note the `println()` at the end of its body, ending that line of the times table
- The second (**inner loop**) iterates over the number of columns
  - Its body prints one entry in the times table

```
for (int rr=1;rr<=rows;rr++) {  
    System.out.print(rr + ": ");  
    for (int cc=1;cc<=cols;cc++) {  
        System.out.print((rr * cc) + " ");  
    }  
    System.out.println();  
}
```

# Nested Loops

The nested loop below prints a triangle of **\*** characters with a height of **h**

- The outer loop iterates over the number of rows
- The inner loop iterates over the number of columns
  - Note that we base the end of the iteration on the value of **ii** the outer loops variable
  - This means we iterate the inner loop a different number of times as we execute

```
for(int ii = 1; ii <= h; ii++) {  
    for(int jj = 1; jj <= ii; jj++) {  
        System.out.print("*");  
    }  
    System.out.println( );  
}
```

```
h=5  
*  
**  
***  
****  
*****
```

# Nested Loops

The example below nests a for loop inside a do-while loop

- The outer loop executes until the user enters a 0
- The inner loop loops prints a list of numbers

```
int times = 0;
do {
    System.out.print("Enter a number (0 to quit): ");
    times = scnr.nextInt();
    for (int ii=0;ii<times;ii++) {
        System.out.print(ii + " ");
    }
    System.out.println();
} while (times > 0);
```

```
Enter a number (0 to quit): 5
0 1 2 3 4
Enter a number (0 to quit): 14
0 1 2 3 4 5 6 7 8 9 10 11 12 13
Enter a number (0 to quit): 8
0 1 2 3 4 5 6 7
Enter a number (0 to quit): 0
```

# Strings & Loops

It is very common to want to loop through the characters of a String one-by-one - we can use a for loop to manage indices of the String

Iterating through a String character-by-character:

```
for(int ii=0; ii<sentence.length(); ii++) {  
    if (sentence.charAt(ii) == ' ') {  
        numSpaces ++;  
    }  
}
```

# Iterators

- An **iterator** is an object that allows you to access elements in a collection one at a time
- This allows you to process each element as needed
- Methods:
  - **hasNext** returns true if there are elements left to process in the iterator, and false otherwise
  - **next** returns the next element in the iterator
- The actual implementation of an iterator is beyond the scope of this course, but if you are interested it is covered in Chapter 9 of the *Java Foundations* book (also covered in CSC205)

# Iterators

- The Java API contains many classes that are iterators
- The **Scanner** class is one example:
  - **hasNext** returns true if there is more data to be scanned
  - **next** returns the next scanned token as a string
  - There are also variations on **hasNext** and **next** that work with specific data types, such as **hasNextInt** & **nextInt**
- Since Scanner objects are iterators, we can use them to
  - Read each line of a file until the end of the file is reached
  - Read each part of a URL to get each part of the path

# The for-each loop

- A variant of the for loop called the **for-each** loop simplifies the repetitive processing for any object that implements the **Iterable** interface
- An **Iterable** interface provides an iterator
- For example, if **BookList** is an **Iterable** object that manages **Book** objects, the following loop will print each book:

```
for (Book myBook : BookList) {  
    System.out.print(myBook) ;  
}
```

- A for-each loop lets us use an iterator without explicitly calling the **hasNext()** and **next()** methods



# The for-each loop

- A variant of the for loop called the **for-each** loop simplifies the repetitive processing for any object that implements the **Iterable** interface
- An **Iterable** interface provides an iterator
- For example, if **BookList** is an **Iterable** object that manages **Book** objects, the following loop will print each book:

```
for (Book myBook : BookList) {  
    System.out.print(myBook) ;  
}
```

- A for-each loop lets us use an iterator without explicitly calling the **hasNext()** and **next()** methods

# A Note About Designing Programs

- You now have quite a few tools to use when developing your programs
  - Variables
  - Conditional Statements
  - Loops
- Programming is hard, even when you are good at it
- One way to simplify programming is **incremental development** - starting your program small and adding one new piece at a time
- Compiling after every 2 or 3 lines (at the most) will help keep you on track
- Test your code once you are done adding each piece
- Remember, a partial program that works is better than a full program that does not run

# Scope

- **Scope** refers to the region of the code that a name is valid in
- Java uses **block scope** meaning that a name is valid from when it is declared through the end of the containing block
- For variables declared inside the code block of a conditional statement or loop, their scope extends to the end of that code block
- If the index of a for is declared in the initialization, then it also only has scope inside the body of the loop
  - However, variables declared outside the loop are still visible when the loop ends, so any changes to them will remain

---

---

**Now go write some code!**

---

---