

# Chapter 4:

# Conditional Execution

---

WEEK 5:

IF/ELSE STATEMENTS

CUMULATIVE ALGORITHMS

TEXT PROCESSING

METHODS WITH CONDITIONAL EXECUTION




# if Statements

---

- if statements give you a way to write code that may not execute **all** of the time
- Whether or not the code within the if block executes will depend on whether it meets some condition

• **Syntax:** `if (<test>){`  
    `<statement>;`  
    `<statement>;`  
    `...`  
    `<statement>;`  
}



- The statements WITHIN the body of the if block are referred to as “**controlled**” statements.
- These statements will only be executed if the “**test**” condition is satisfied


# if/else Statements

---

- The if/else statement is a variation that provides two alternatives and that executes one or the other
- If the test condition is not met, the else block code executes instead of the if block code

- **Syntax:**

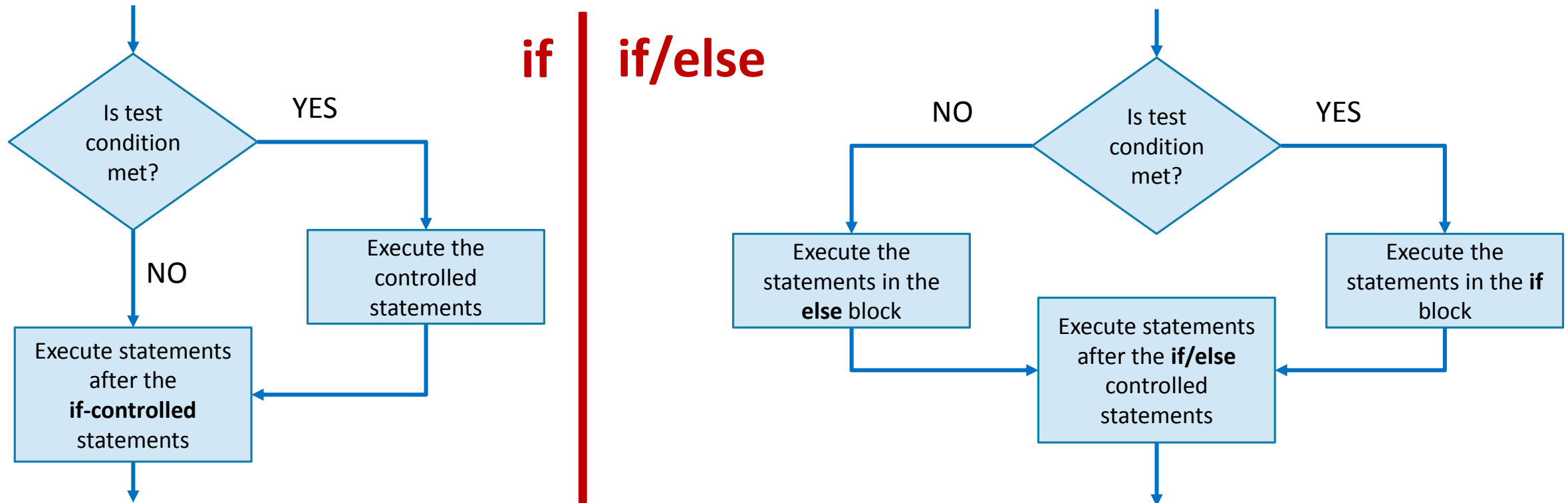
```
if (<test>){  
    <statement>;  
    <statement>;  
    ...  
    <statement>;  
}  
else {  
    <statement>;  
}
```



- If the test condition is not met, the statements in the if block are not executed.
- Instead, control jumps to the else block, and the code statements in that block are executed

# Control Structures: if and if/else statements

- Like the for loop, the if and if/else statements are **CONTROL STRUCTURES**



# Relational Operators

---

- As described, the if and if/else statements are controlled by **tests**
- tests are **expressions** that have the following format

**<expression> <relational operator> < expression>**

Operator	Meaning	Example	Value
==	Equal to	2 + 2 == 4	true
!=	Not equal to	3.2 != 4.1	true
<	Less than	4 < 2	false
>	Greater than	4 > 3	true
<=	Less than or equal to	2 <= 0	false
>=	Greater than or equal to	2.4 >= 1.6	true

# Precedence of Relational Operators

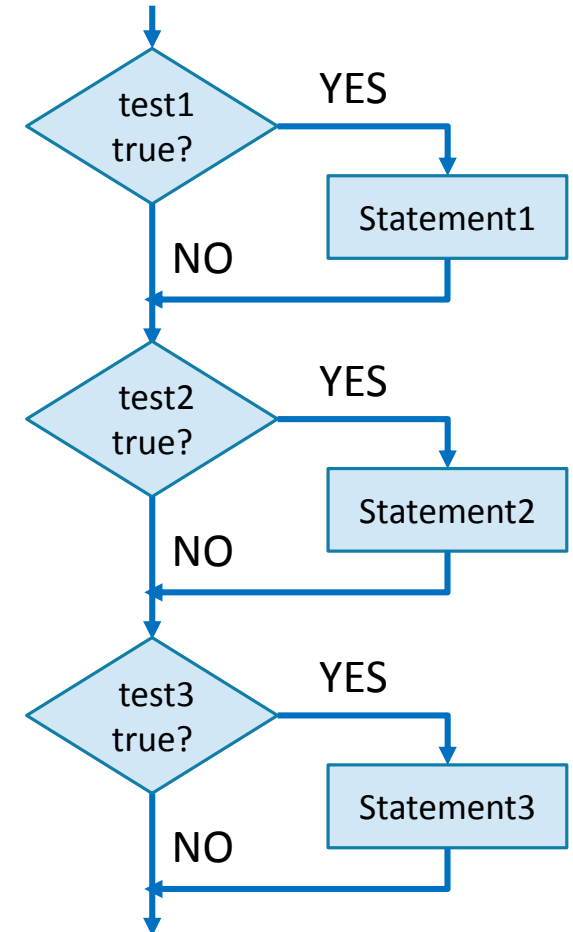
---

- Relational operators have **LOWER PRECEDENCE** than arithmetic operators
- This means that the expressions on **BOTH SIDES** of the relational operator will be evaluated as separate expressions **BEFORE** the two sides are compared using the relational operator
- **In other words, Java will perform all the “math” operations before it tests any relationships**
- **Note:** technically, “==” has a higher precedence than the other relational operators, but all relational operators are lower precedence than the arithmetic operators

# Sequential if Statements

- if statements can occur in series, one after the other
- When this occurs, each test is applied separately from the others
- Potentially, ALL the if test conditions could satisfy their respective tests, leading to execution of ALL the if-controlled statements
- This differs from the situation that occurs when the if statements are included as elements of nested if/else statements (shown on next slide)

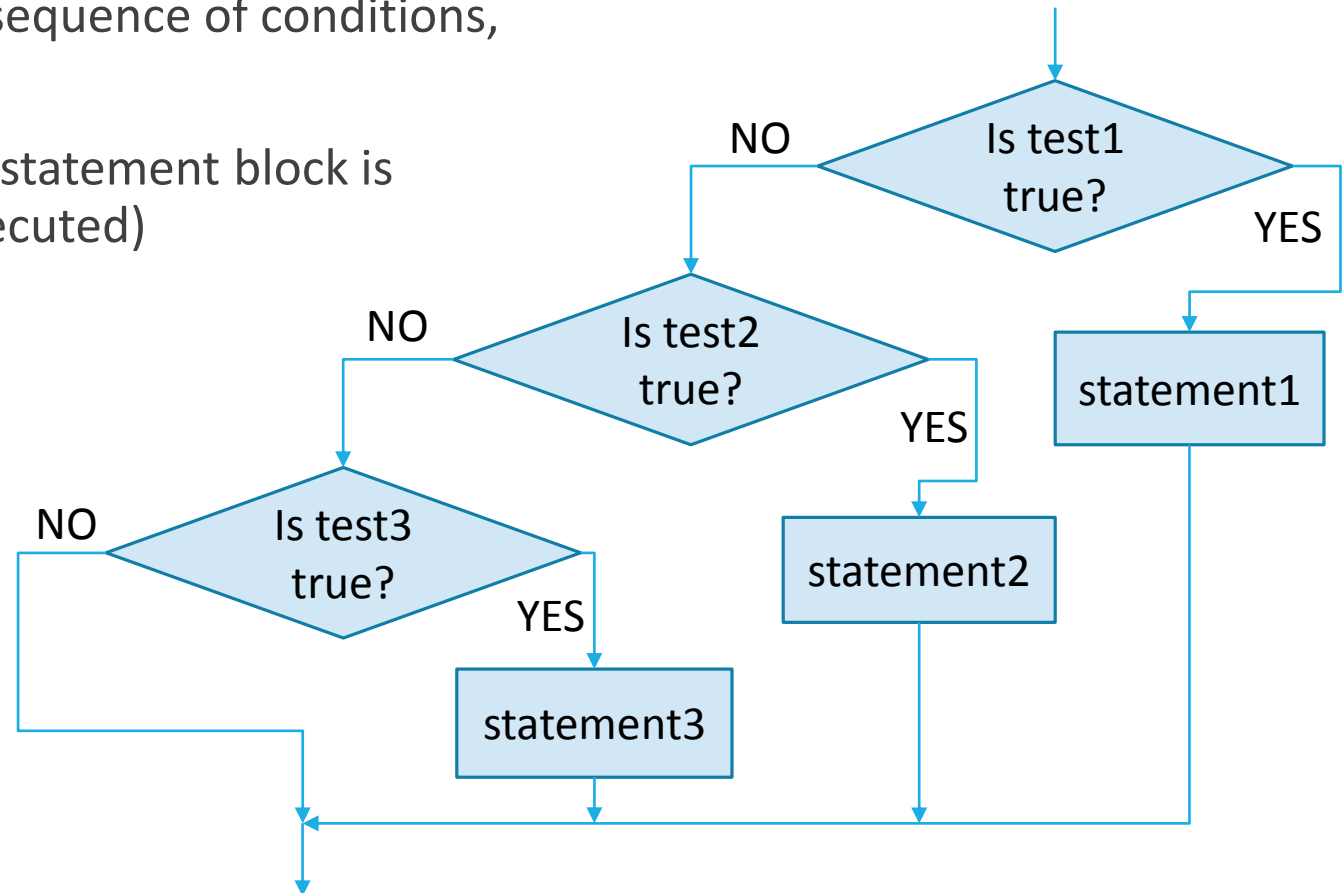
```
if (<test1>){  
    <statementBlock1>;  
}  
if (<test2>){  
    <statementBlock2>;  
}  
if(<test3>{  
    <statementBlock3>;  
}
```



# Nested if/else Statements

- Nested if/else statements can be used to check a sequence of conditions, stopping once **ANY** of them is met
- When if/else statements are nested, at most **ONE** statement block is executed (if no tests are satisfied, none may be executed)
- **Syntax:**

```
if (<test1>) {  
    <statementBlock1>;  
} else if(<test2>) {  
    <statementBlock2>;  
} else if(<test3>) {  
    <statementBlock3>;  
}
```

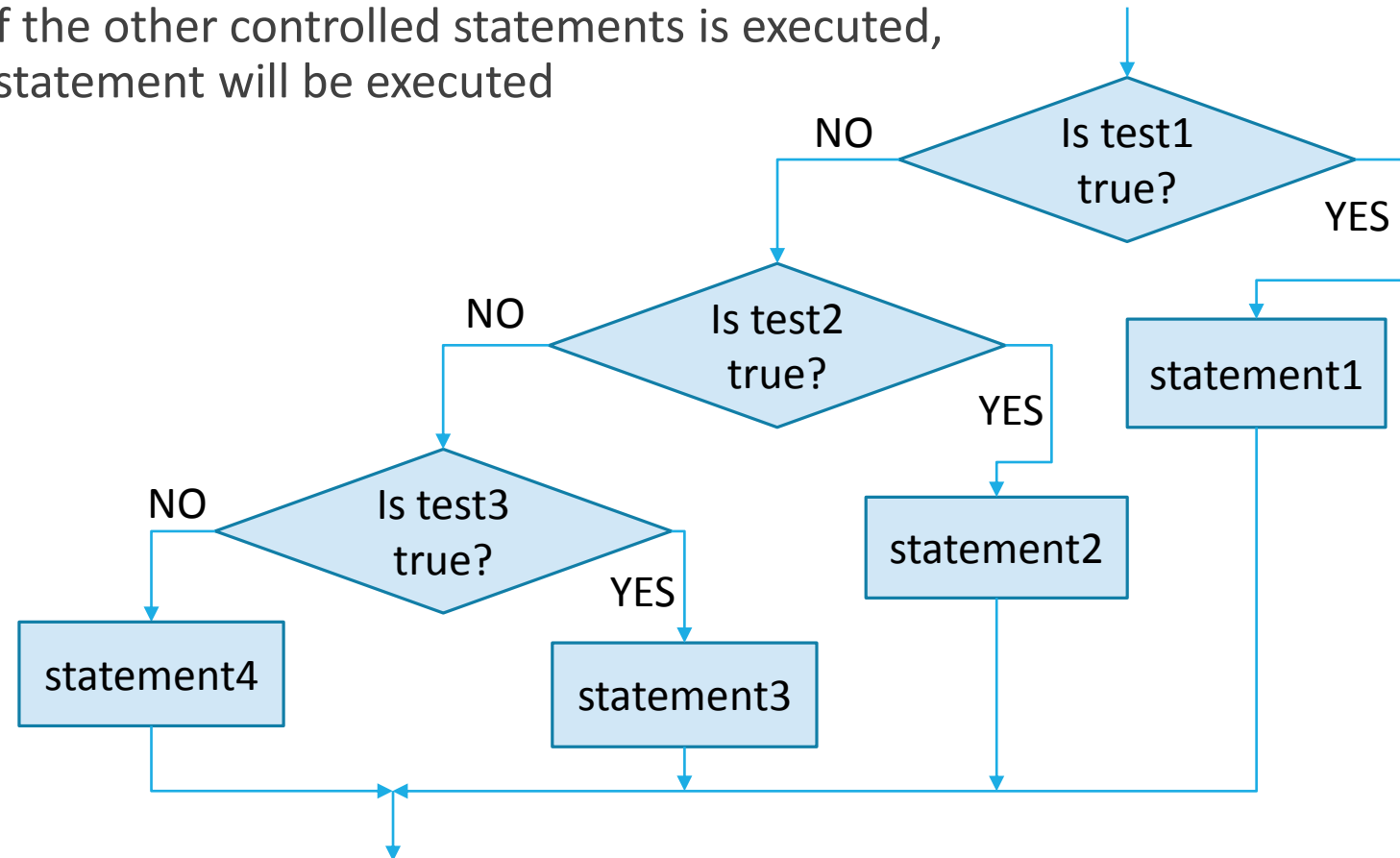




# Final else after nested if/else statements

- When a final **else** block is included, if none of the other controlled statements is executed, the block of code controlled by the final else statement will be executed

```
if (<test1>) {  
    <statementBlock1>;  
} else if(<test2>) {  
    <statementBlock2>;  
} else if(<test3>) {  
    <statementBlock3>;  
} else {  
    <statementBlock4>;  
}
```



# if/else Options

Situation	Construct	Basic Form
You want to execute any combination of controlled statements	Sequential ifs (each block has its own test that determines whether or not it gets executed)	<pre>if (&lt;test1&gt;){     &lt;statement1&gt;; } if (&lt;test2&gt;){     &lt;statement2&gt;; }</pre>
You want to execute zero or one of the controlled statements	Nested ifs ending in test	<pre>if (&lt;test1&gt;){     &lt;statement1&gt;; } else if (&lt;test2&gt;){     &lt;statement2&gt;; }</pre>
You want to execute exactly one of the controlled statements	Nested ifs ending in else	<pre>if (&lt;test1&gt;){     &lt;statement1&gt;; } else if (&lt;test2&gt;){     &lt;statement2&gt;; } else{     &lt;statement3&gt;; }</pre>

# Object Equality

---

- You have seen how == and != can be used to test for equality and nonequality of PRIMITIVE data
- These operators SHOULD NOT BE USED for objects such as strings (in general – see note below)
- There is a different way to test whether objects are “equal”
- Every Java object has a method called equals that takes another object as an argument
- This topic will be covered further in Chapter 8

**NOTE:** Java is not always consistent about how String comparisons work. Using the equals method with String objects will always work correctly. However, *some* versions of Java may allow String objects to be compared with “==” (as if they were primitives) in some cases.

**I would recommend using the equals method** EVEN IF you have a version that seems to allow “==”, as the “==” method may not work in all situations (e.g. if one of the strings being compared is obtained via the console versus created in the code).

# Object Equality with String Objects

## Code Examples:

```
System.out.print("Enter the single letter y (lower case) to signify yes: ");
String response = console.next(); // Fetch a single token

if(response == "y"){
    System.out.println("Your choice was: yes");
} else{
    System.out.println("Your choice was NOT yes");
}
System.out.println("**Is the output correct?");
```

```
if(response.equals("y")){
    System.out.println("Your choice was: yes");
} else{
    System.out.println("Your choice was NOT yes");
}
System.out.println("**Is the output correct NOW?");
```

```
System.out.println("But Java is letting me use == in the test below:");
String a = "yes", b = "yes";
if(a == b)
    System.out.println("a is the same as b");
```

## Outputs:

```
Enter the single letter y (lower case) to signify yes: y
Your choice was NOT yes
**Is the output correct?
```

```
Your choice was: yes
**Is the output correct NOW?
```

```
But Java is letting me use == in the test below:
a is the same as b
```

# Testing Multiple Conditions

---

- We can combine tests using logical operators
- For example, perhaps you asked the user of your program to input the month value as a number
- You might want to check that the input provided is a valid input value
- You could do this as shown:

```
if (input >= 1 && input <= 12){  
    <statements>  
} else {  
    <probably want to ask for input again>  
}
```

- The code above uses the **logical AND operator** ( **&&** ) to check that both conditions are satisfied
- You can also use the **logical OR operator** ( **||** ) to check if one condition OR ANOTHER is satisfied
- This will be covered further in Chapter 5

# Cumulative Algorithms

---

- Cumulative algorithms incrementally compute some overall value, often by using a loop

## **pseudocode example 1:**

### **(to find sum of a list of numbers):**

set sum = 0 (outside the loop)

for (all values in the list):

    update the sum to equal sum + current list value

display final sum

## **pseudocode example 2**

### **(to find the maximum value**

### **in a list of numbers):**

initialize max -- either to the lowest possible value or to the first value (outside the loop)

for (all values in list):

    compare each value to the current max

    if value > max:

        reset current max to equal value

Display max value

# More on Cumulative Algorithms

---

## **pseudocode example 3:**

### **(finding the minimum of a list of numbers):**

set min to max possible value or to first value  
(outside the loop)

for (all values in the list):

    compare each value to the current min

    if value < min

        reset current min to equal value

display min value

## **Cumulative sums using an if statement**

Useful for keeping track of how many items meet  
some preset condition

- Counting occurrence of letters in a word
- Counting occurrence of negative numbers in a list of numbers

# Roundoff Errors

---

- Roundoff errors are numerical errors that occur because floating point numbers are stored as approximations rather than as exact values
- floating point numbers are stored in a format similar to scientific notation, with a set of digits and an exponent
- Problems occur when the number being stored can't be expressed exactly using base 2
- If storing exact values is critical, it is best to store the values as type int
- when comparing two floating point numbers, tests of equality can be problematic – instead, use a test such as to see if the difference in the two numbers you're considering is less than some (small) preset threshold amount
- The **abs() method** (absolute value) in the math library can be useful when carrying out such comparisons



# Text Processing

---

- **Text Processing**: the editing and formatting of strings of text
- Programming problems often involve creating, editing, examining, and formatting text
- The **char** primitive type represents a single **character** of text
- A literal value of type char is expressed by enclosing the character within **single quotes**:

```
char ch = 'A';
```

- It is also legal to create a char value that represents an escape sequence:

```
char newline = '\n';
```

# Differences between char Primitives & String Objects

---

- A **char** occupies a **very small amount of memory** and has **NO METHODS**
- a single character **can** be stored as a String object, but at times using a **char** can be simpler

# char vs int

---

- Values of type **char** are stored internally as 16-bit **INTEGERS**
- Unicode is a standard encoding scheme that determines which integer values are used to represent each character
- Since characters are “really” integers, Java can understand (and evaluate) expressions such as:  
**char letter = 'a' + 2; // this command stores the value 'c' (as an integer) in the variable letter**
- An **int** can be converted to a **char** by using a type case (this is required because not ALL integers represent characters)

# Examples using char and int

---

```
public static void main(String[] args) {  
  
    char letter = 'a';  
    System.out.println(letter);  
  
    int code = letter; //Java automatically converts type char to int when int is expected  
    System.out.println(code);  
  
    char letterAgain = (char)code; //int converted to char using a type cast  
    System.out.println(letterAgain);  
  
    System.out.println("The result of letter + 2 is: " + (letter + 2));  
    // Java converted letter to int to do the addition  
  
    char newLetter = (char) (letter + 2);  
    System.out.println("The result of adding 2 to letter now is: " + newLetter);  
  
    char newestLetter = 'f' + 5;  
    System.out.println("the result of \'f\' + 5 is: " + newestLetter);  
}
```

## OUTPUT

```
<terminated> charVsIntDemo [Java Application] C:\Program  
a  
97  
a  
The result of letter + 2 is: 99  
The result of adding 2 to letter now is: c  
the result of 'f' + 5 is: k
```

# Cumulative Text Algorithms

---

- Strings of characters are often used in **cumulative algorithms** (algorithms that incrementally update a given value)
- For example, you might want to **count** the number of times a particular letter occurs in a string
- You might also want to use a loop to gradually **build up** a string
  - This can be accomplished by **concatenating** a **char** value to a **string** by using the **+ operator**
  - This can be useful for tasks such as reversing a string

# Formatting string output

---

- So far, we've used *System.out.print* and *System.out.println* for console output
- Java has a third method: ***System.out.printf***
- *System.out.printf* gives us more control over the format of our output (but is a bit more complicated to use)
- **What can you do with *System.out.printf*?**
  - Print out sequences of numbers neatly aligned in vertical columns (tabular appearance)
  - Print output with a specified number of places after the decimal point (useful for displaying monetary data)

# Format Strings

---

- The System.out.printf method accepts a specially written string called a **format string**
- The format string specifies the general appearance of the output, and is followed by any parameters to be included in the output

- **Syntax:**

**System.out.printf(<format string>, <parameter>, ..., <parameter>);**

- A format string is similar to a regular string, except that it can contain **PLACEHOLDERS** called **format specifiers**
- format specifiers start with a % sign and end with a letter specifying the kind of value that will eventually occupy the spot
- Since format specifiers start with a “%”, if you want to print out an actual percentage sign, you need to specify this by typing TWO percentage signs (%%) in a row

# Examples using Format Strings

- Format specifiers are more than just simple placeholders – they also contain information on the width, precision, and alignment of the value being printed
- Note use of “\n” at the end of the text strings (printf does not advance to the next line the way println does)

```
public class formatStringDemo {  
    public static void main(String[] args) {  
        int x = 38, y = -152;  
        System.out.printf("location: (%d, %d)\n", x, y);  
  
        int score = 87;  
        System.out.printf("You got %d%% on the exam.\n", score);  
  
        double price = 5.487839;  
        System.out.printf("the price of that item is $%.2f\n", price );  
    }  
}
```

## OUTPUT:

```
<terminated> formatStringDemo [Java Application] !  
location: (38, -152)  
You got 87% on the exam.  
the price of that item is $5.49
```



# Methods with Conditional Execution

---

- At the beginning of this unit, we introduced conditional execution
- When programming, you should generally break a complex task into smaller subtasks
- According, your programs will generally consist of a number of smaller methods joined together, with each method accomplishing some specific task that contributes to solving the larger problem
- These methods need to work well together – this is easier to accomplish if it is clear what inputs each method is expecting and what outputs it will produce
- This information can be provided in the form of **preconditions** and **postconditions**
- **Precondition:** A condition that must be true before a method executes in order to guarantee that the method can perform its task.
- **Postcondition:** A condition that the method guarantees will be true after it finishes executing, as long as the preconditions were true before the method was called.

# Throwing Exceptions

---

- When an error occurs in the execution of a program, it is useful to have the program provide some information about the failure
- Some exceptions are built into Java, for example `ArithmeticException`, `IllegalArgumentException`

For a full list, see: [http://www.tutorialspoint.com/java/java\\_built\\_in\\_exceptions.htm](http://www.tutorialspoint.com/java/java_built_in_exceptions.htm)

- Other times, however, you want to specify when exceptions should be generated in your own code
- The Java classes that cover certain types of exceptions can be used to provide customized “error messages” that give the user of your program with helpful feedback as to what went wrong
- Constructing these exception objects is especially helpful in the case of preconditions failing to be met and including these exceptions is an example of defensive programming

# Revisiting Return Values

---

- As control structures, if and if/else statements can be used to select what values your methods will return from a set of possible options
- For example, you can write a method to return the maximum of two numbers
  - Depending on the actual values of the numbers input in to the method, sometimes the first number will be the maximum and sometimes the second number will be
  - If the two numbers are equal, it doesn't matter which one is classified as the "maximum"
- The following code shows how to return the appropriate selection every time the method executes:

**NOTE:** Only 1 return statement will be executed, since as soon as a method returns a value, control is restored to the point following the method call.

```
public static double getMax(double a, double b){  
    //check if first number is larger (or equal) and return it if it is  
    if(a >= b){  
        return a;  
    }  
    return b;    //otherwise, return the second number  
}
```

# indexOf Method

---

- The Java String class has a method `indexOf` that returns the index of the first occurrence of the letter passed in to the method
- This method is an example of an instance method, because it is one of the methods that “belongs to” the String class
- What if you wanted to write a static method to accomplish the same thing?
- A static method does not “belong to” any particular class – consequently, you would need to pass both the letter to search for AND the string to search in as arguments to the method
- Since the method needs to return an int value, this must be specified in the method header:

**`public static int indexOf(char letter, String searchString)`**

# indexOf Method, cont.

---

- To find the character we are looking for, we should iterate through the characters in the string
- When we find the specified character, the method should return the index of the location where it was found
- If the letter we're searching for is NOT in the string we are searching, our method should return an "impossible" value, such as -1, to indicate this
- **REMEMBER:** Once the method returns a value, flow control returns to the point in the program where the method was called (just after the method call)
- Thus, if the letter is located while in the loop iterating through the characters of the searchString, the code after the loop will never be executed

# Code for Static indexOf Method

```
public class example {  
    public static void main(String[] args) {  
        String teststring = "This is my test string -- I will search for z and see if it shows up.";  
        int result = indexOf('z', teststring);  
        System.out.printf("The letter z was found at index %d of the string.", result);  
        System.out.println();  
        int result2 = indexOf('q', teststring);  
        System.out.printf("The letter q was found at index %d of the string.", result2);  
    }  
    public static int indexOf(char letter, String searchString){  
        for (int i = 0; i < searchString.length(); i++){  
            if (searchString.charAt(i) == letter){  
                return i;  
            }  
        }  
        return -1;  
    }  
}
```

## OUTPUT:

```
<terminated> example [Java Application] C:\Program Files\Java\jre  
The letter z was found at index 44 of the string.  
The letter q was found at index -1 of the string.
```

**if statement** used to test for the letter we're searching for – the value of the current index will only be returned if the letter is there. If the letters occurs more than once in the string, only the first occurrence will be found by this method.

# Concluding Comments

---

- **if** and **if/else** statements give you a way to write code that may not execute **all** of the time
- **if** and **if/else** statements must include a **test** to determine which “branch” gets executed
- These tests generally often contain **relational operators** that perform comparisons
- The relational operator **==** works primarily on **PRIMITIVES**
- If comparing **objects**, use the **equals method**: `objectA.equals(objectB)`
- **Roundoff errors** can cause issues when comparing **floats** – use `abs()` to find the (positive) difference between the 2 values being compared (eg: `abs(float1 – float2)` ) and compare this value to some pre-set “acceptable tolerance”
- **char** values are stored as integers – Java will treat them as integers when `int` type is expected and `int` values which correspond to `char` values may be converted using a type cast
- **printf** gives us control over how output is displayed (width, alignment, precision)