INTERACTIVE PROGRAM

Interactive Programs

- Programs generally need input on which to operate
- The `Scanner` class provides convenient methods for reading input values of various types
- A `Scanner` object can be set up to read input from various sources, including the user typing values on the keyboard
- Keyboard input is represented by the `System.in` object

Reading Input

- The following line creates a Scanner object that reads from the keyboard:

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

- The `new` operator creates the `Scanner` object
- Once created, the `Scanner` object can be used to invoke various input methods, such as:

```
answer = scan.nextLine();
```

Reading Input

- The `Scanner` class is part of the `java.util` class library, and must be imported into a program to be used
    - You need to import the java.util library

```
import java.util.Scanner;
public class TestScanner{
        public static void main(String[] args){
                Scanner scan = new Scanner (System.in);
        }
}
```

Reading Input

- The `nextLine` method reads all of the input until the end of the line is found
- Unless specified otherwise, white space is used to separate the elements (called tokens) of the input
- White space includes space characters, tabs, new line characters
- The `next` method of the Scanner class reads the next input token and returns it as a string
- Methods such as `nextBoolean, nextByte, nextShort, nextInt, nextLong, nextFloat and nextDouble` read data of particular types

Scanner Example

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

boolean bo = sc.nextBoolean();

byte b = sc.nextByte();
short s = sc.nextShort();
int x = sc.nextInt();
long l = sc.nextLong();

double a = sc.nextDouble();
float f = sc.nextFloat();

String st1= sc.nextLine();
//the rest of the current line， excluding any line separator at the end
String st2= sc.next();
//returns the next complete token
```

STRING

String

- There are only eight primitive type in Java – `boolean, byte, short, int, long, float, double, and char`
- The variables of eight primitive type called *primitive variables*
- String ***is not* a primitive type** in Java, but `String` is a class name which can be used as a type to declare an *object variable*
- An *object variable* holds the address of an object

String Class

```
public final class String{
        …
        …
        public int length(){
                return count;.
        }
        public boolean isEmpty(){
                return count == 0;
        }
        public char charAt(int index){
                if ((index < 0) || (index >= count)){
                        throw new StringIndexOutOfBoundsException(index);
                }
                return value[index + offset];
        }
        …
        …
}
```

String Objects

- Because strings are so common, Java provides two ways to create String objects

1. Creating String object by using the `new` operator is called *instantiation*

   ```
   String course= new String("Java Programming");
   ```

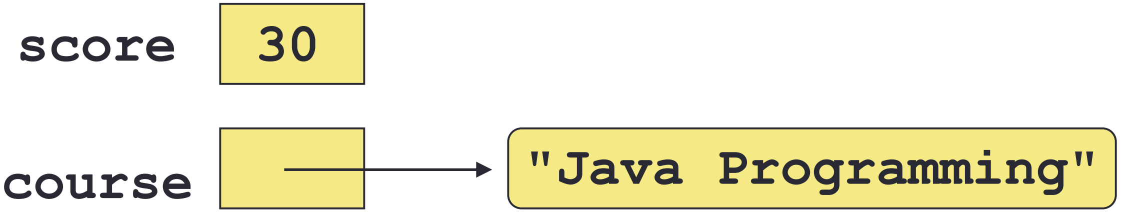
2. Enclosing your character string within double quotes will automatically create a new String object

   ```
         String course = "Java Programming";
   ```

   This is special syntax that works only for strings

References

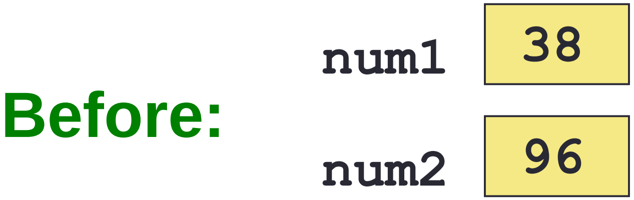
- Note that a primitive variable contains the value itself, but an object variable contains the address of the object
- An object reference can be thought of as a pointer to the location of the object



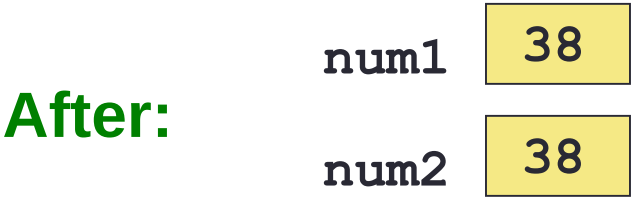
Primitive Assignment

- The act of assignment takes a copy of a value and stores it in a variable

- For primitive types:



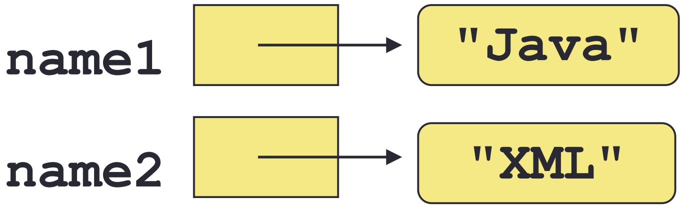
```
num2 = num1;
```



Reference Assignment

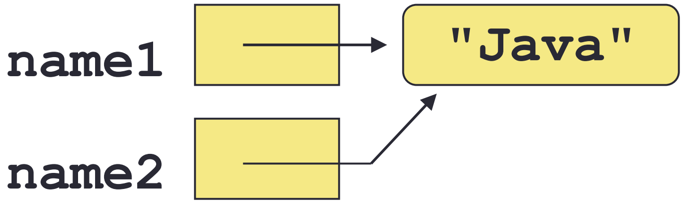
- For object references, assignment copies the address:

Before:



```
name2 = name1;
```

After:



Aliases

- Two or more references that refer to the same object are called *aliases* of each other
- That creates an interesting situation: one object can be accessed using multiple reference variables
- Aliases can be useful, but should be managed carefully
- Changing an object through one reference changes it for all of its aliases, because there is really only one object

Garbage Collection

- When an object no longer has any valid references to it, it can no longer be accessed by the program
- The object is useless, and therefore is called *garbage*
- Java performs *automatic garbage collection* periodically, returning an object's memory to the system for future use
- In other languages, the programmer is responsible for performing garbage collection

Invoking Methods

- Once String object has been created, we can use the *dot operator* to invoke its methods

```
count = course.length()
```

- A method may *return a value*, which can be used in an assignment or expression
- A method invocation can be thought of as asking an object to perform a service

String Methods

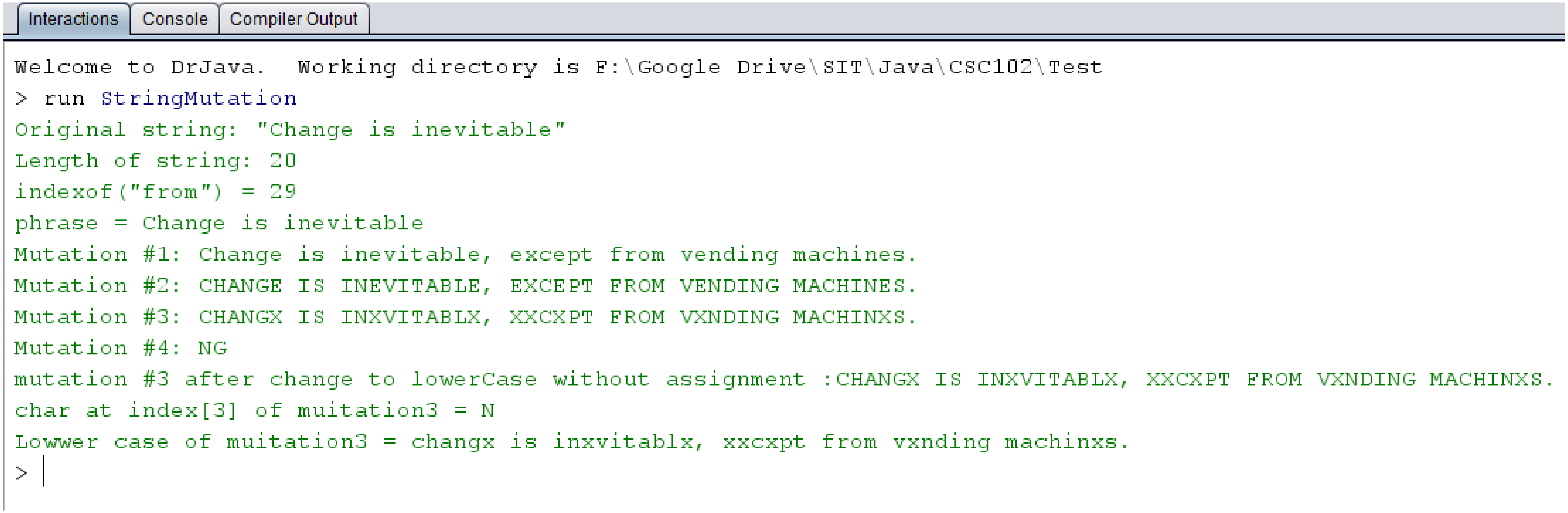
- Once a `String` object has been created, neither its value nor its length can be changed
- Thus we say that an object of the `String` class is *immutable*
- However, several methods of the `String` class return new `String` objects that are modified versions of the original

String Methods

```
String phrase = "Change is inevitable";
String mutation1, mutation2, mutation3, mutation4;

System.out.println ("Original string: \"" + phrase + "\"");
System.out.println ("Length of string: " + phrase.length());

mutation1 = phrase.concat (", except from vending machines.");
System.out.println("indexof(\"from\") = "+ mutation1.indexOf("from"));
System.out.println("phrase = "+phrase);
mutation2 = mutation1.toUpperCase();
mutation3 = mutation2.replace ('E', 'X');
mutation4 = mutation3.substring (3, 5);

// Print each mutated string
System.out.println ("Mutation #1: " + mutation1);
System.out.println ("Mutation #2: " + mutation2);
System.out.println ("Mutation #3: " + mutation3);
System.out.println ("Mutation #4: " + mutation4);

mutation3.toLowerCase();
System.out.println("mutation #3 after change to lowerCase without assignment
:"+mutation3);
System.out.println("char at index[3] of muitation3 = "+mutation3.charAt(3));
System.out.println(“Lower case of muitation3 = "+mutation3.toLowerCase());
```

String Methods

- The given program print the following result



```
Welcome to DrJava.  Working directory is F:\Google Drive\SIT\Java\CSC102\Test
> run StringMutation
Original string: "Change is inevitable"
Length of string: 20
indexof("from") = 29
phrase = Change is inevitable
Mutation #1: Change is inevitable, except from vending machines.
Mutation #2: CHANGE IS INEVITABLE, EXCEPT FROM VENDING MACHINES.
Mutation #3: CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
Mutation #4: NG
mutation #3 after change to lowerCase without assignment :CHANGX IS INXVITABLX, XXCXPT FROM VXNDING MACHINXS.
char at index[3] of muitation3 = N
Lowwer case of muitation3 = changx is inxvitablx, xxcxpt from vxnding machinxs.
> 
```

String Indexes

- It is occasionally helpful to refer to a particular character within a string
- This can be done by specifying the character's numeric *index*
- The indexes begin at zero in each string
- In the string `"Hello"`, the character `'H'` is at index 0 and the `'o'` is at index 4

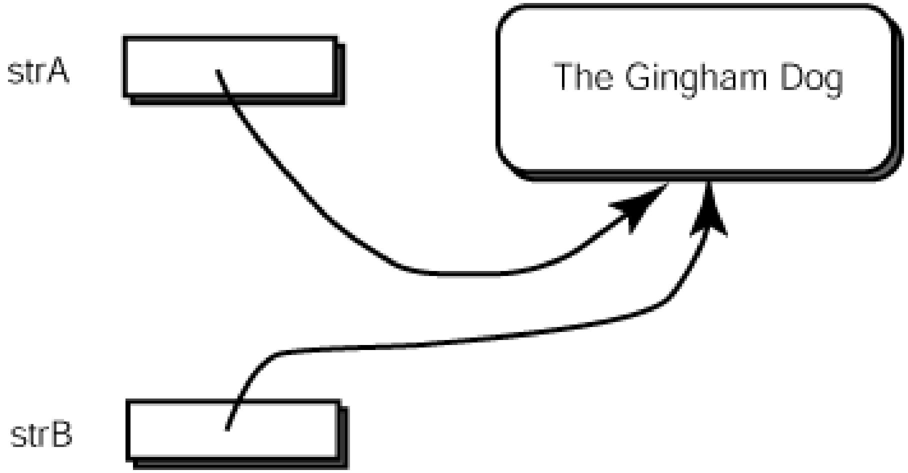
Equality of Reference Variable Contents

- The == operator looks at the contents of two reference variables.
- If both reference variables contain the same reference, then the result is true. Otherwise the result is false.
- The == operator does NOT look at objects! It only looks at references (information about where an object is located).

```
String strA = new String( "The Gingham Dog" );
String strB = new String( "The Calico Cat" );
if (strA == strB)
    System.out.println( “strA and strB point to the same object.");
```

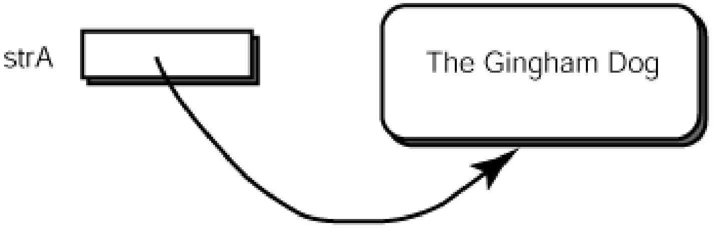
Two Reference Variables Pointing to One Object.

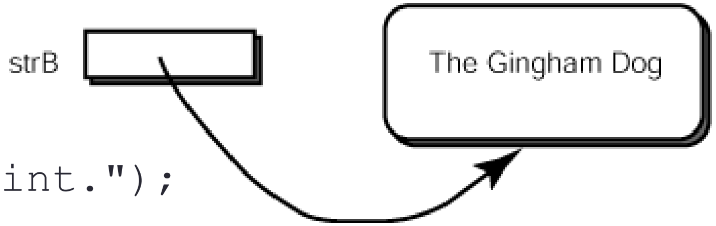
```
String strA;   // reference to the object
String strB;   // another reference to the object

strA = new String( "The Gingham Dog" );

System.out.println (strA);

strB = strA;

System.out.println (strA);
System.out.println (strB);

if ( strA == strB )
   System.out.println ("Same info in each reference variable.");
```



Two Objects with Equivalent Contents

```
class EgString6 {
  public static void main ( String[] args )  {
    String strA;   // reference to the first object
    String strB;   // reference to the second object

    strA = new String("The Gingham Dog");

    System.out.println(strA );

    strB = new String("The Gingham Dog");
    System.out.println(strB);

    if (strA == strB)
      System.out.println("This will not print.");
  }
}
```





The equals() Method

- The equals(String) method of class String tests if two Strings contain the same characters.
- The equals(String) method looks at objects. It detects equivalence.
- The == operator detects identity. For example,

```
String strA;   // first object
String strB;   // second object

strA = new String("The Gingham Dog");
strB = new String("The Gingham Dog");

if (strA.equals(strB))
  System.out.println("This WILL print.");

if (strA == strB)
  System.out.println("This will NOT print.");
```

Other String Methods

- `public char` **`charAt`**`(int index)`
- `public String` **`concat`**`(String str)`
- `public boolean` **`equalsIgnoreCase`**`(String anotherString)`
- `public int` **`length`**`()`
- `public int` **`indexOf`**`(String str)`
- `public int` **`indexOf`**`(String str, int fromIndex)`
- `public String` **`substring`**`(int beginIndex)`
- `public String` **`substring`**`(int beginIndex, int endIndex)`

DIY – Explore String Methods

- Write a program using the string methods introduced in the previous slide
- In your program, print the result of each method and use comment to explain what does the method do
- Example

```
String st1 = new String("Java Programming");
String st2 = "Java Programming";
System.out.println(st1.concat(st2));
//concat(String inputstring) method is used to join the string
with the input string.
//In the example, it join String st1 with String st2.
```

SELECTION

Control Flow and Control Structure

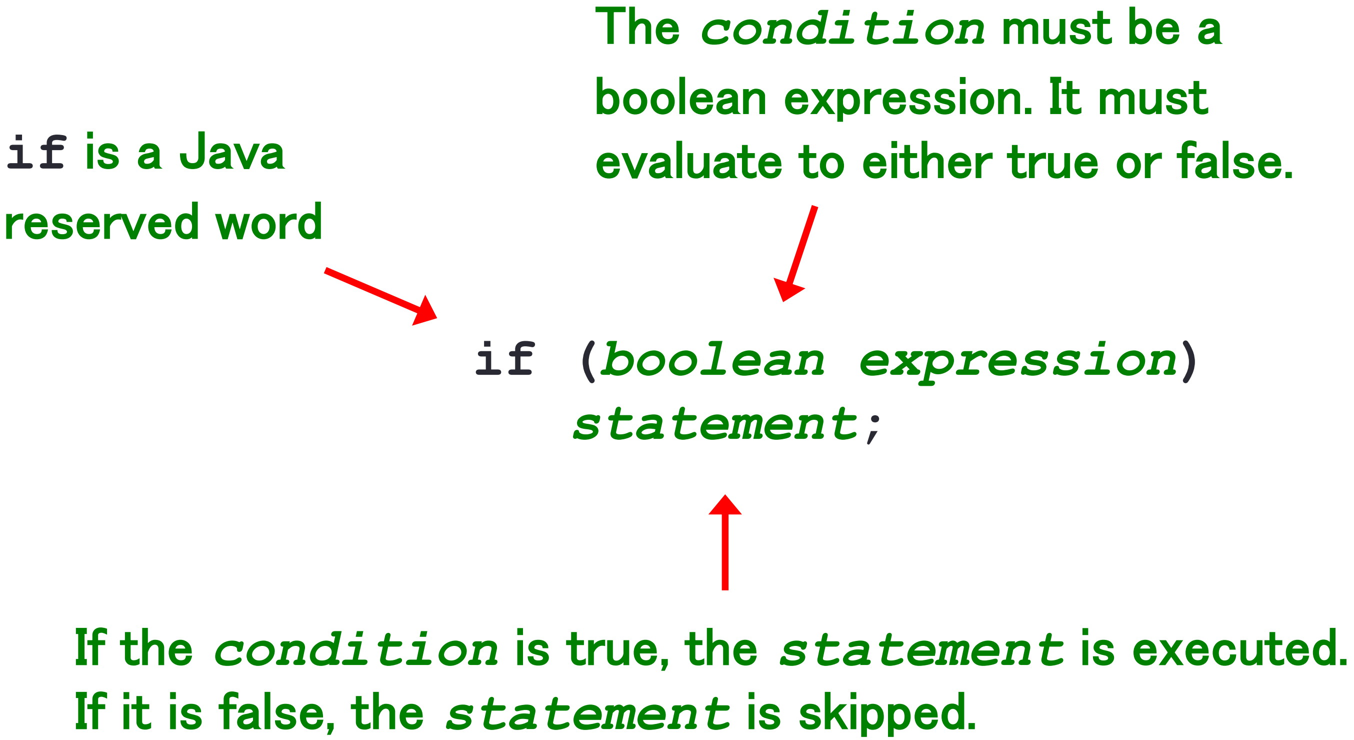
- The order in which a program's statements execute is called its control flow.
- A programmer specifies a program's control flow.
- Control Structure
  - Sequence logic structure
  - **Selection (Branch) logic structure**
  - Repetition (Loop) logic structure

Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
- Therefore they are sometimes called *selection statements*
- Conditional statements give us the power to make basic decisions
- The Java conditional statements are the:
  - if statement
  - if-else statement
  - conditional operators
  - switch statement

The if Statement

- The *if statement* has the following syntax:



`if` is a Java reserved word

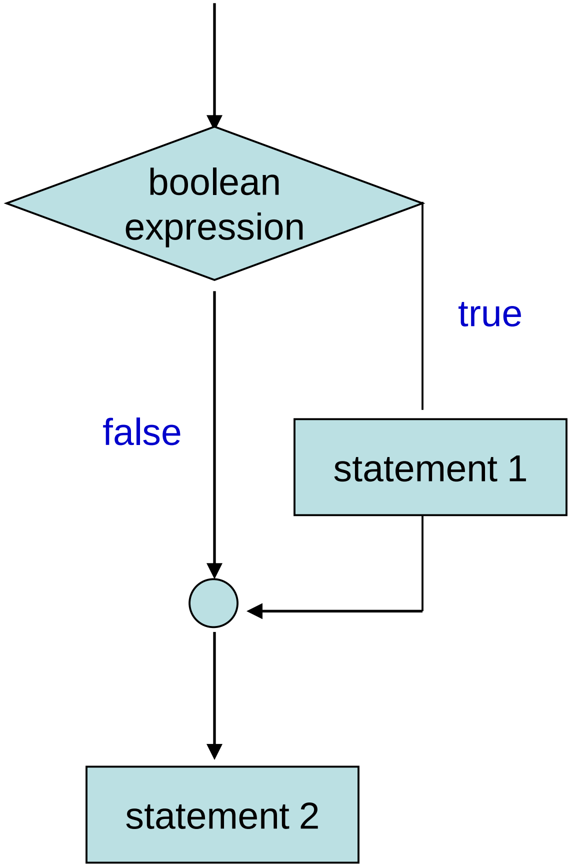
The *`condition`* must be a boolean expression. It must evaluate to either true or false.

```
if (boolean expression)
    statement;
```

If the *`condition`* is true, the *`statement`* is executed.
If it is false, the *`statement`* is skipped.

Logic of an if statement

```
if (boolean expression)
    statement1;
Statement2
```



Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

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

- Note the difference between the equality operator (==) and the assignment operator (=)

The if Statement

- An example of an `if` statement:

```
if (sum > MAX)
   delta = sum - MAX;
System.out.println ("The sum is " + sum);
```

First the condition is evaluated -- the value of `sum` is either greater than the value of `MAX`, or it is not

If the condition is true, the assignment statement is executed -- if it isn’t, it is skipped.

Either way, the call to `println` is executed next

The if Statement

- What do the following statements do?

```
if (top >= MAXIMUM)
   top = 0;
```

Sets `top` to zero if the current value of `top` is greater than or equal to the value of `MAXIMUM`

```
if (total != stock + warehouse)
   inventoryError = true;
```

Sets a flag to true if the value of `total` is not equal to the sum of `stock` and `warehouse`

The precedence of the arithmetic operators is higher than the precedence of the equality and relational operators

The if Statement

- What does this program do?

```
final int MINOR = 21;

Scanner scan = new Scanner (System.in);

System.out.print ("Enter your age: ");
int age = scan.nextInt();

System.out.println ("You entered: " + age);

if (age < MINOR)
        System.out.println ("Youth is a wonderful thing. Enjoy.");
System.out.println ("Age is a state of mind.");
```

Indentation

- The statement controlled by the `if` statement is indented to indicate that relationship
- The use of a consistent indentation style makes a program easier to read and understand
- Although it makes no difference to the compiler, proper indentation is crucial

Indentation Revisited

- Remember that indentation is for the human reader, and is ignored by the computer

```
if (total > MAX)
   System.out.println ("Error!!");
   errorCount++;
```

Despite what is implied by the indentation, the increment will occur whether the condition is true or not

if Block Statements

- Several statements can be grouped together into a *block statement* delimited by braces

- A block statement can be used wherever a statement is called for in the Java syntax rules

```
if (total > MAX){
   System.out.println ("Error!!");
   errorCount++;
}
```

Logical Operators

- Boolean expressions can also use the following *logical operators*:

!	**Logical NOT**
&	**Logical AND (&& short-circuited AND)**
|	**Logical OR (|| short-circuited OR)**

- They all take boolean operands and produce boolean results
- Logical NOT is a unary operator (it operates on one operand)
- Logical AND and logical OR are binary operators (each operates on two operands)

Logical NOT

- The *logical NOT* operation is also called *logical negation* or *logical complement*
- If some boolean condition `a` is true, then `!a` is false; if `a` is false, then `!a` is true
- Logical expressions can be shown using a *truth table*

`a`	`!a`
true	false
false	true

Logical AND and Logical OR

- The *logical AND* expression

  ```
  a && b
  ```

  is true if both `a` and `b` are true, and false otherwise

- The *logical OR* expression

  ```
  a || b
  ```

  is true if `a` or `b` or both are true, and false otherwise

Logical Operators

- Expressions that use logical operators can form complex conditions

```
if (total < MAX+5 && !found)
   System.out.println ("Processing…");
```

All logical operators have lower precedence than the relational operators

Logical NOT has higher precedence than logical AND and logical OR

Logical Operators

- A truth table shows all possible true-false combinations of the terms
- Since `&&` and `||` each have two operands, there are four possible combinations of conditions `a` and `b`

`a`	`b`	`a && b`	`a || b`
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Boolean Expressions

- Specific expressions can be evaluated using truth tables

total < MAX	found	!found	total < MAX && !found
false	false	true	false
false	true	false	false
true	false	true	true
true	true	false	false

Short-Circuited Operators

- The processing of logical AND and logical OR is “short-circuited”
- If the left operand is sufficient to determine the result, the right operand is not evaluated

```
if (count != 0 && total/count > MAX)
   System.out.println ("Testing…");
```

This type of processing must be used carefully

The & and | Operators

- If x is 1, what is x after this expression?

```
(x > 1) & (x++ < 10)
(x > 1) && (x++ < 10)
```

- If x is 1, what is x after this expression?

```
(1 > x) && ( 1 > x++)
(1 > x) & ( 1 > x++)
```

- How about

```
(1 == x) || (10 > x++)
(1 == x) | (10 > x++)
```

The if-else Statement

- An *else clause* can be added to an `if` statement to make an *if-else statement*

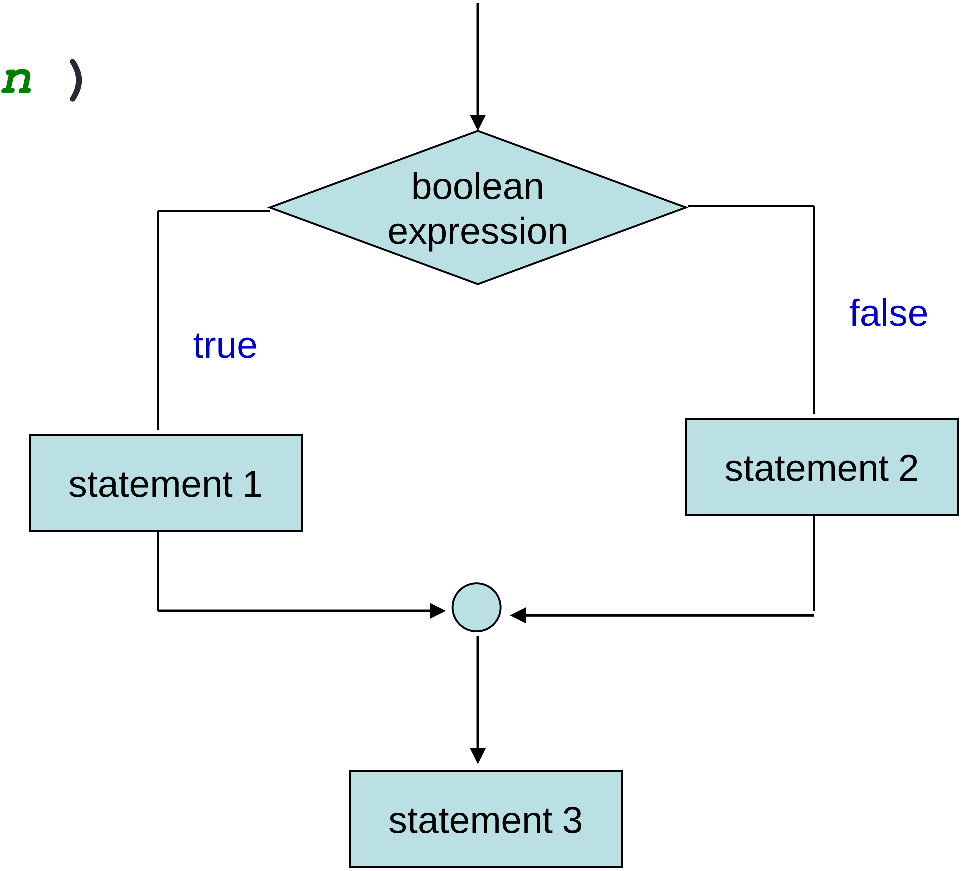
```
if (boolean expression )
    statement1;
else
    statement2;
```

If the *`condition`* is true, *`statement1`* is executed; if the condition is false, *`statement2`* is executed

One or the other will be executed, but not both

Logic of an if-else statement

```
if (boolean expression )
    statement1;
else
    statement2;
statement3
```



if-else Block Statements

- In an `if-else` statement, the `if` portion, or the `else` portion, or both, could be block statements

```
if (total > MAX){
   System.out.println ("Error!!");
   errorCount++;
}
else{
   System.out.println ("Total: " + total);
   current = total*2;
}
```

if-else Block Statements

```
if (temp<37)
    System.out.println(“You are healthy”);
else
    System.out.println(“You may be sick”);
    System.out.println(“You should see a doctor”);
```

Not part of the else clause

```
if (temp<37)
    System.out.println(“You are healthy”);
else{
    System.out.println(“You may be sick”);
    System.out.println(“You should see a doctor”);
}
```

Use block to prevent misleading

The Conditional Operator

- Java has a *conditional operator* that uses a boolean condition to determine which of two expressions is evaluated

- Its syntax is:

```
condition ? expression1 : expression2
```

- If the `condition` is true, `expression1` is evaluated; if it is false, `expression2` is evaluated

- The value of the entire conditional operator is the value of the selected expression

The Conditional Operator

- The conditional operator is similar to an `if-else` statement, except that it is an expression that returns a value

- For example:

```
larger = ((num1 > num2) ? num1 : num2);
```

- If `num1` is greater than `num2`, then `num1` is assigned to `larger`; otherwise, `num2` is assigned to `larger`

- The conditional operator is *ternary* because it requires three operands

The Conditional Operator

- Another example:

```
System.out.println ("Your change is " + count +
   ((count == 1) ? "Dime" : "Dimes"));
```

If `count` equals 1, [redacted]

If `count` is anything other than 1, [redacted]

Nested if Statements

- The statement executed as a result of an `if` statement or `else` clause could be another `if` statement
- These are called *nested if statements*
- An `else` clause is matched to the last unmatched `if` (no matter what the indentation implies)
- Braces can be used to specify the `if` statement to which an `else` clause belongs

Multiple Alternative if Statements

```
if (score >= 90)
   grade = ‘A’;
else
   if (score >= 80)
      grade = ‘B’;
   else
      if (score >= 70)
         grade = ‘C’;
      else
         if (score >= 60)
            grade = ‘D’;
         else
            grade = ‘F’;
```

Dangling Else

```
if(c1)
    if(c2)
         s1;
else
         s2;
```

Incorrect

```
if(c1)
    if(c2)
         s1;
    else
         s2;
```

Correct

Dangling Else

- The else clause matches the most recent if clause in the same block. For example, the following statement

```
int i = 1; int j = 2; int k = 3;
if (i > j)
  if (i > k)
    System.out.println("A");
else
  System.out.println("B");
```

- is equivalent to

```
int i = 1; int j = 2; int k = 3;
if (i > j)
  if (i > k)
    System.out.println("A");
  else
    System.out.println("B");
```

Dangling Else

- Nothing is printed from the preceding statement. To force the else clause to match the first if clause, you must add a pair of braces:

```
int i = 1;
int j = 2;
int k = 3;
  if (i > j) {
    if (i > k)
      System.out.println("A");
  }
  else
    System.out.println("B");
```

- This statement prints B.

Multi-way if-else Statement

```
if (score >= 90)
   grade = ‘A’;
else
   if (score >= 80)
      grade = ‘B’;
   else
      if (score >= 70)
         grade = ‘C’;
      else
         if (score >= 60)
            grade = ‘D’;
         else
            grade = ‘F’;
```

Equivalent

```
if (score >= 90)
   grade = ‘A’;
else if (score >= 80)
   grade = ‘B’;
else if (score >= 70)
   grade = ‘C’;
else if (score >= 60)
   grade = ‘D’;
else
   grade = ‘F’;
```

DIY – if-else Statement

- Write a program that takes a number input from System.in and assign to variable *y*. Then write an if statement that assigns 5 to variable *x* if *y* is greater than or equal 100. Otherwise, *x* is equal to 10.

The switch Statement

- The *switch statement* provides another way to decide which statement to execute next
- The `switch` statement evaluates an expression, then attempts to match the result to one of several possible *cases*
- Each case contains a value and a list of statements
- The flow of control transfers to statement associated with the first case value that matches

The switch Statement

- The general syntax of a `switch` statement is:

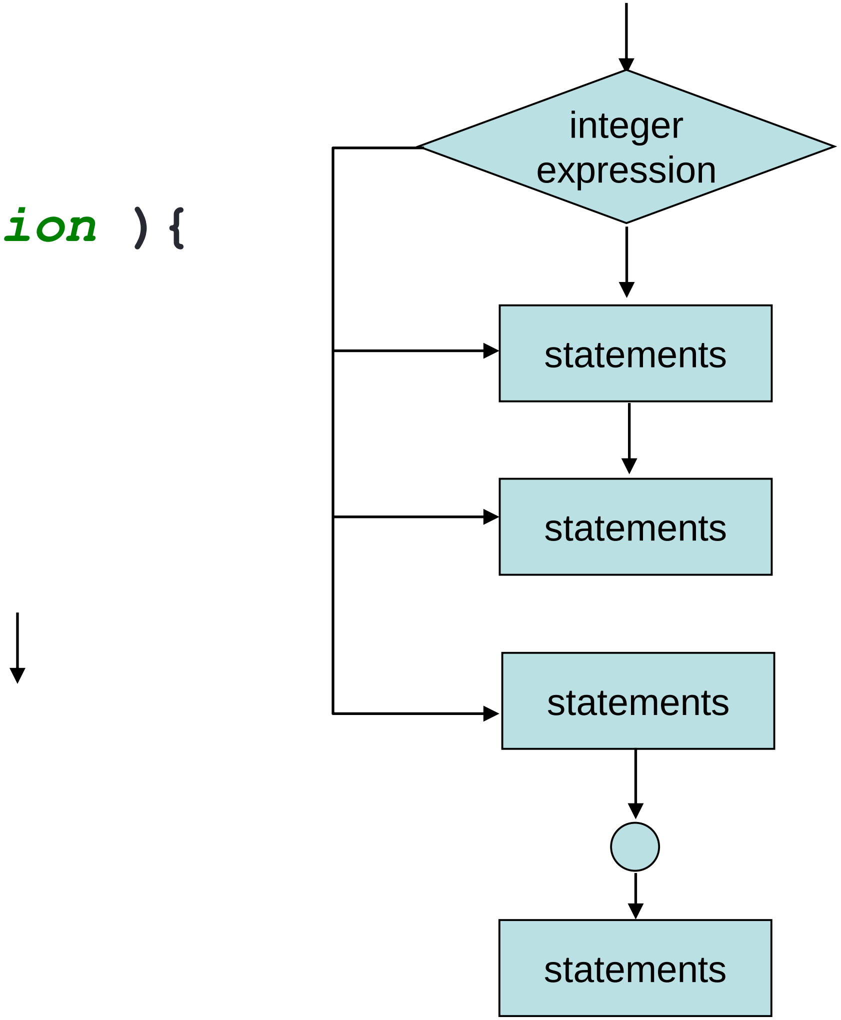
`switch` **and** `case` **are reserved words**

```
switch (integer expression ){
    case value1 :
        statement-list1
    case value2 :
        statement-list2
    case value3 :
        statement-list3
    case  ...

}
```

If *expression* matches *value2*, control jumps to here

Switch Statement

```
switch (integer expression ){
    case value1 :
        statement-list1
    case value2 :
        statement-list2
    case value3 :
        statement-list3
    case  ...

}
```



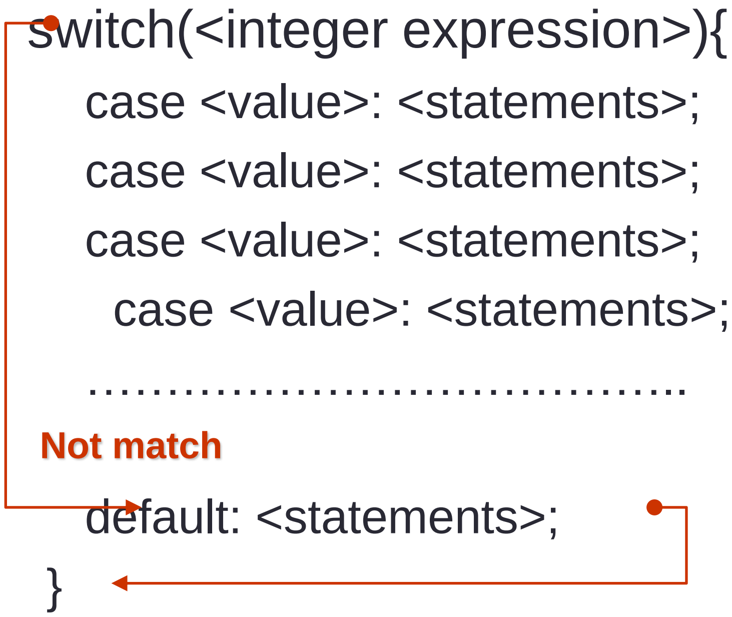
The switch Statement

- The expression of a `switch` statement must result in an *integral type*, meaning an integer (`byte`, `short`, `int`) or a `char`
- In version 1.7 up, switch expression can be String type
- It cannot be a `boolean` value or a floating point value (`float` or `double`) or `long`
- The implicit boolean condition in a `switch` statement is equality
- You cannot perform relational checks with a `switch` statement

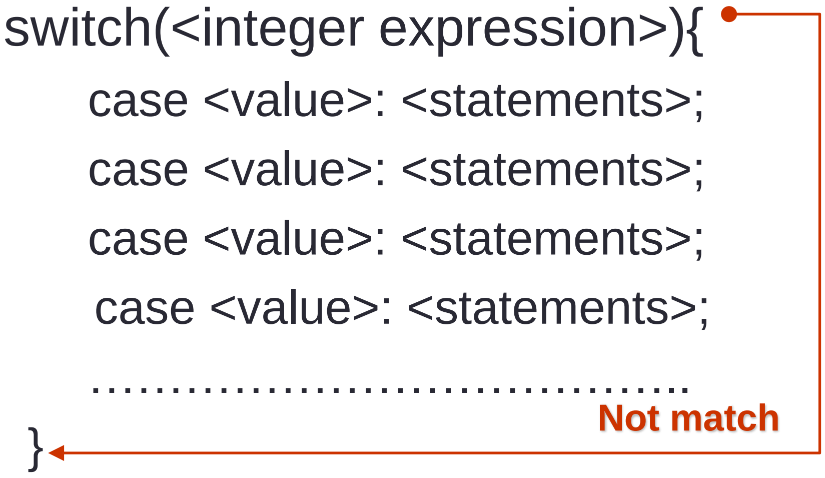
The switch Statement

- A `switch` statement can have an optional *default case*
- The default case has no associated value and simply uses the reserved word `default`
- If the default case is present, control will transfer to it if no other case value matches
- If there is no default case, and no other value matches, control falls through to the statement after the switch

Default Case



```
switch(<integer expression>){
    case <value>: <statements>;
    case <value>: <statements>;
    case <value>: <statements>;
      case <value>: <statements>;
    …………………………………..
Not match
    default: <statements>;
}
```

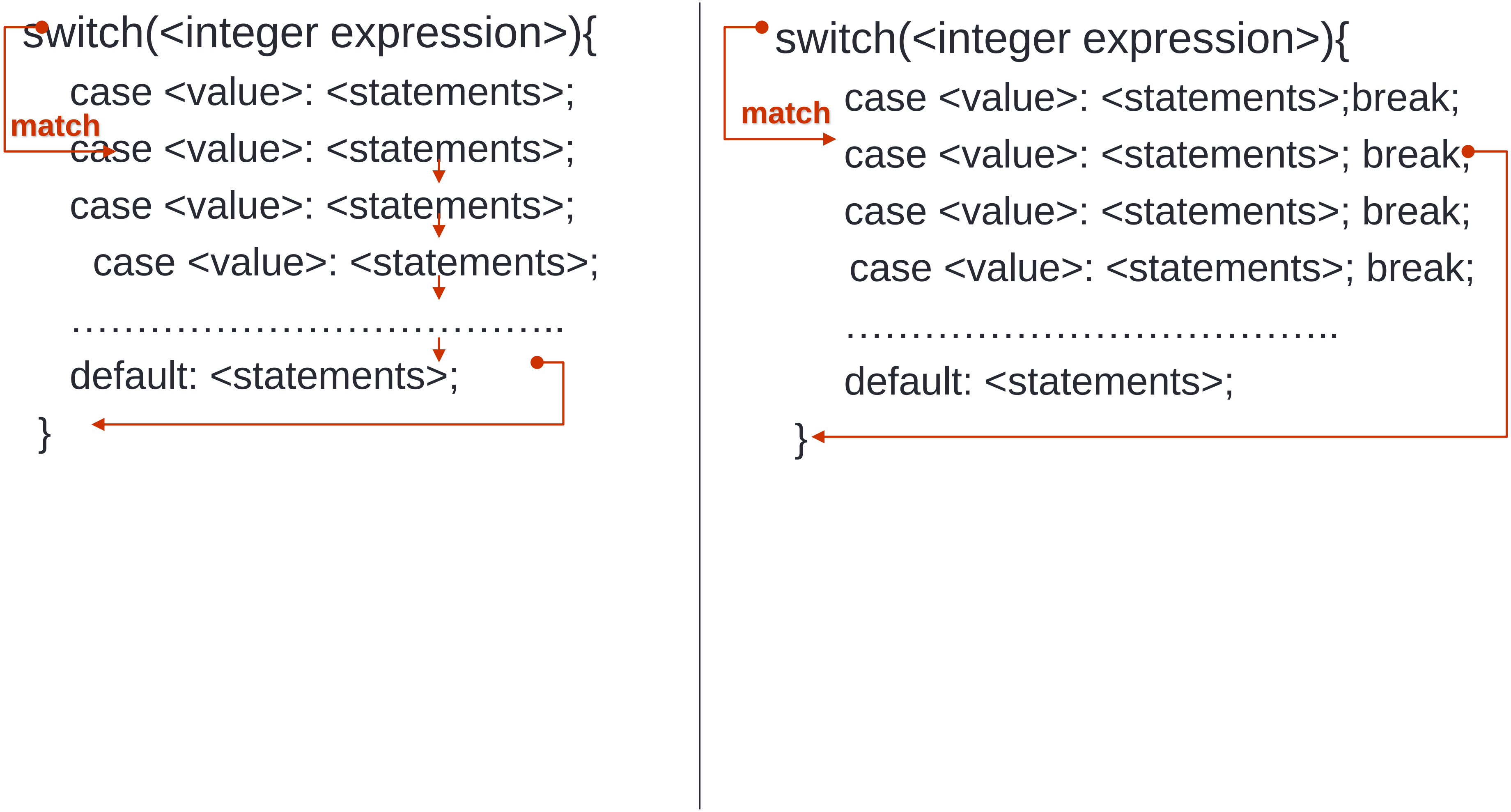


```
switch(<integer expression>){
      case <value>: <statements>;
      case <value>: <statements>;
      case <value>: <statements>;
       case <value>: <statements>;
      …………………………………..
                                 Not match
  }
```

The switch Statement

- Often a *break statement* is used as the last statement in each case's statement list
- A `break` statement causes control to transfer to the end of the `switch` statement
- If a `break` statement is not used, the flow of control will continue into the next case
- Sometimes this may be appropriate, but often we want to execute only the statements associated with one case

Switch Statement



```
switch(<integer expression>){
    case <value>: <statements>;
    case <value>: <statements>;
    case <value>: <statements>;
      case <value>: <statements>;
    .....................................
    default: <statements>;
}
```

```
switch(<integer expression>){
    case <value>: <statements>;break;
    case <value>: <statements>; break;
    case <value>: <statements>; break;
     case <value>: <statements>; break;
    .....................................
    default: <statements>;
  }
```

The switch Statement

- An example of a switch statement:

```
switch (option){
   case 'A':
      aCount++;
      break;
   case 'B':
      bCount++;
      break;
   case 'C':
      cCount++;
      break;
}
```

Comparing Data

- When comparing data using boolean expressions, it's important to understand the nuances of certain data types
- Let's examine some key situations:
  - Comparing floating point values for equality
  - Comparing characters
  - Comparing string

Comparing Float Values

- You should rarely use the equality operator (`==`) when comparing two floating point values (`float` or `double`)
- Two floating point values are equal only if their underlying binary representations match exactly
- Computations often result in slight differences that may be irrelevant
- In many situations, you might consider two floating point numbers to be "close enough" even if they aren't exactly equal

Comparing Float Values

- To determine the equality of two floats, you may want to use the following technique:

```
if (Math.abs(f1 - f2) < TOLERANCE)
   System.out.println ("Essentially equal");
```

If the difference between the two floating point values is less than the tolerance, they are considered to be equal

The tolerance could be set to any appropriate level, such as 0.000001

Danger with Comparing Doubles by using ==

- Floating point arithmetic is not exact

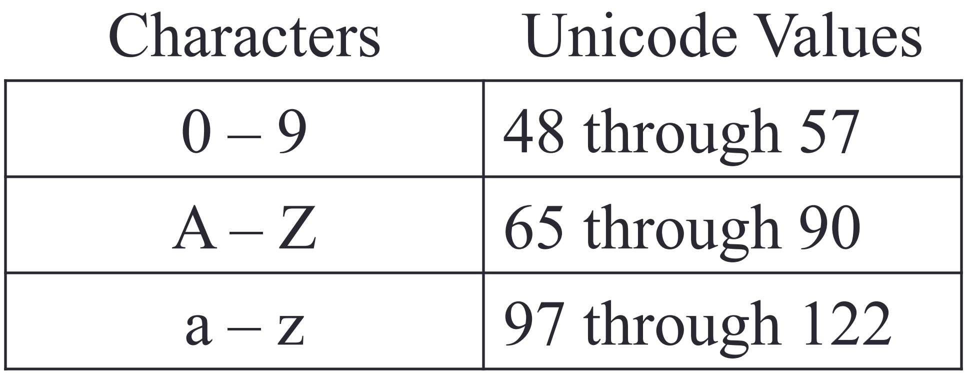
```
class DecimalFraction {
  public static void main (String[] args)  {
    float x = 1.0f;    // 1.0f means 1.0 float
    float y = 10.0f;

    if ( x/y == 0.1 )
          System.out.println("x/y == 0.1”);
    else
          System.out.println("x/y != 0.1");
  }
}
```

Comparing Characters

- Java character data is based on the Unicode character set
- Unicode establishes a particular numeric value for each character, and therefore an ordering
- We can use relational operators on character data based on this ordering
- For example, the character `'+'` is less than the character `'J'` because it comes before it in the Unicode character set

Comparing Characters

- In Unicode, the digit characters (0-9) are contiguous and in order
- Likewise, the uppercase letters (A-Z) and lowercase letters (a-z) are contiguous and in order



Characters	Unicode Values
0 – 9	48 through 57
A – Z	65 through 90
a – z	97 through 122

Comparing Characters

```
char ch1='B';
char ch2='a';
System.out.println(ch1<ch2); //true
System.out.println(ch1>ch2); //false
System.out.println(ch1==ch2); //false
```

Comparing Strings

- Remember that in Java a character string is an object
- The `equals` method can be called with strings to determine if two strings contain exactly the same characters in the same order
- The `equals` method returns a boolean result

```
if (name1.equals(name2))
   System.out.println ("Same name");
```

Comparing Strings

- We cannot use the relational operators to compare strings
- The `String` class contains a method called `compareTo` to determine if one string comes before another
- A call to `name1.compareTo(name2)`
  - returns zero if `name1` and `name2` are equal (contain the same characters)
  - returns a negative value if `name1` is less than `name2`
  - returns a positive value if `name1` is greater than `name2`

Comparing Strings

```
if (name1.compareTo(name2) < 0)
   System.out.println (name1 + "comes first");
else
   if (name1.compareTo(name2) == 0)
      System.out.println ("Same name");
   else
      System.out.println (name2 + "comes first");
```

Because comparing characters and strings is based on a character set, it is called a *lexicographic ordering*

Lexicographic Ordering

- Lexicographic ordering is not strictly alphabetical when uppercase and lowercase characters are mixed
- For example, the string `"Great"` comes before the string `"fantastic"` because all of the uppercase letters come before all of the lowercase letters in Unicode
- Also, short strings come before longer strings with the same prefix (lexicographically)
- Therefore `"book"` comes before `"bookcase"`