Lecture 02: Back to basics

CS 0445: Data Structures

Constantinos Costa

http://db.cs.pitt.edu/courses/cs0445/current.term/

Sep 5, 2019, 8:00-9:15
University of Pittsburgh, Pittsburgh, PA

Objects and Classes

- An object : a program construct that contains data and can perform certain actions
  – Objects interact with one another to accomplish a particular task
- Actions performed by objects are defined by methods in the program
  – Valued methods return a value
  – Void methods do not

Identifiers

- Use identifiers to name certain parts of a program
  - Consists entirely of letters, digits, the underscore character _, and the dollar sign $
  - Cannot start with a digit, must not contain a space or any other special character
- Java is case sensitive

Identifiers

- Common practice
  - Start the names of classes with uppercase letters
  - Start the names of objects, methods, and variables with lowercase letters

Reserved Words

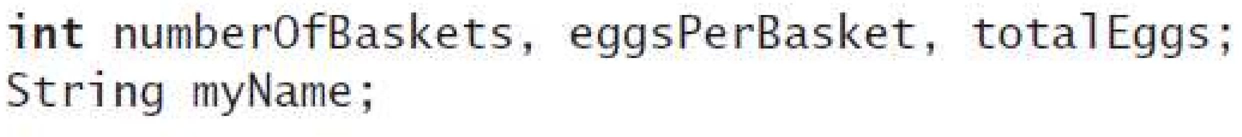
- Some words have a special predefined meaning in Java
  - Also called keywords
  - Cannot use these words for variable names
  - Used only for the intended purpose

Variables

- Represents a memory location that stores data such as numbers and letters
  - Number or letters stored there are the *value*
  - That value can be changed
- The variable's data type specifies what kind of value may be stored
  - Primitive type
  - Reference type
  - Class type
  - Array type

Variables

- Variable declaration indicates the type of data the variable will hold
  - Write a type name
  - Followed by a list of variable names separated by commas
  - Ending with a semicolon



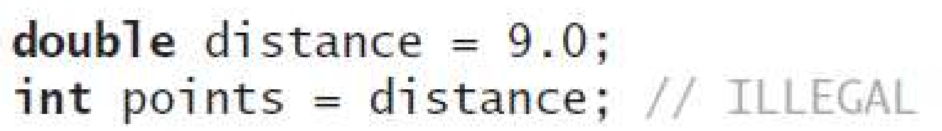
```
int numberOfBaskets, eggsPerBasket, totalEggs;
String myName;
```

Primitive Types

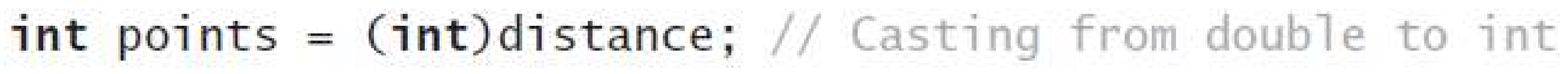
- Integers
  - Byte, int, short, long
- Floating point
  - Float, double
- Char (single chartacters)

Type Casting

- Changing of the type of a value to some other type
- Note the *wrong* and *right* way to do this



```
double distance = 9.0;
int points = distance; // ILLEGAL
```

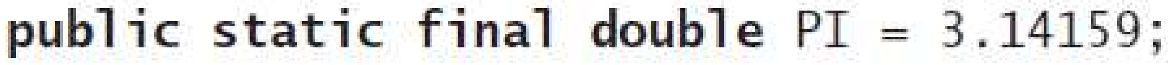


```
int points = (int)distance; // Casting from double to int
```

Named Constants

- Mechanism allows you to define and initialize a variable *and* fix the variable's value
  - Thus, it cannot be changed

- Good practice to place named constants
  - Ne

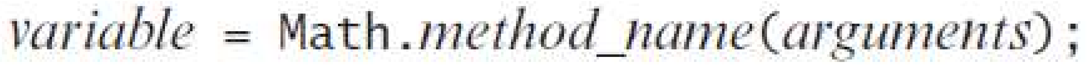


```
public static final double PI = 3.14159;
```

  - Outside of any method definitions.
- Typically use all uppercase for named constant

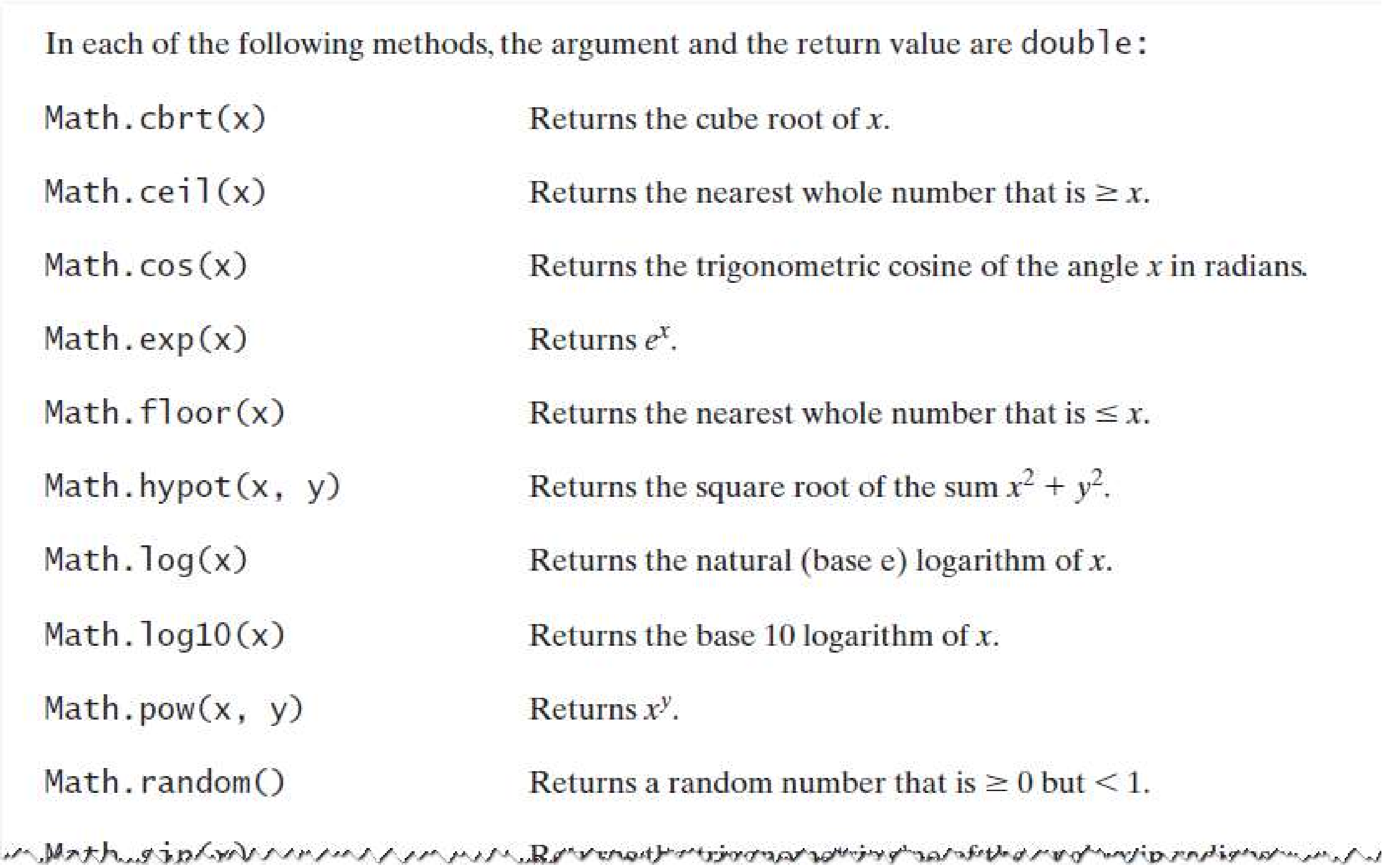
The Class **Math**

- Provides a number of standard mathematical methods.
  - Static methods
  - Write the class name, a dot, the name of the method, and a pair of parentheses
  - Most **Math** methods require that you specify items within the pair of parentheses



```
variable = Math.method_name(arguments);
```

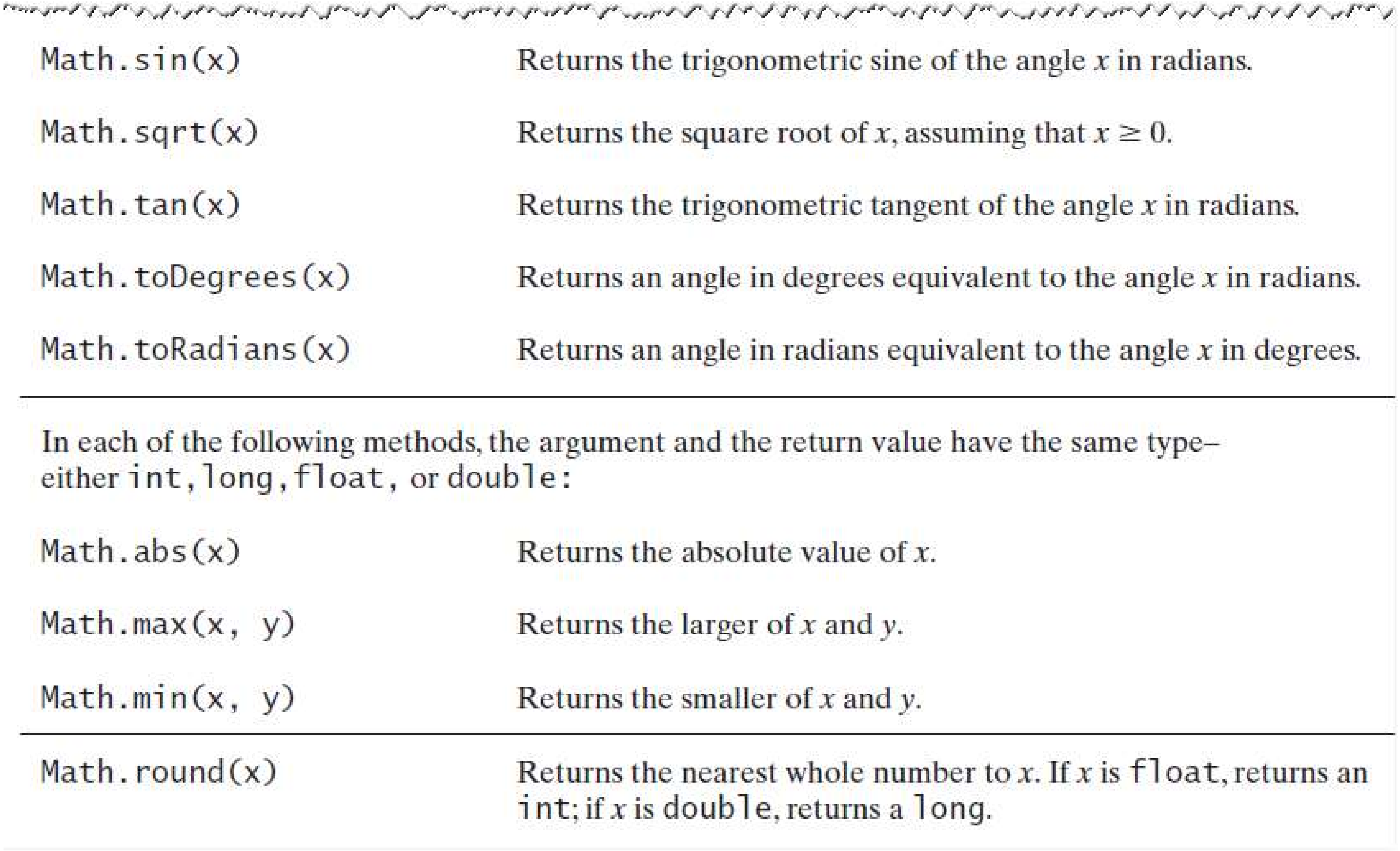
The Class **Math**



In each of the following methods, the argument and the return value are `double`:

`Math.cbrt(x)`	Returns the cube root of x.
`Math.ceil(x)`	Returns the nearest whole number that is $\geq x$.
`Math.cos(x)`	Returns the trigonometric cosine of the angle x in radians.
`Math.exp(x)`	Returns e^x.
`Math.floor(x)`	Returns the nearest whole number that is $\leq x$.
`Math.hypot(x, y)`	Returns the square root of the sum $x^2 + y^2$.
`Math.log(x)`	Returns the natural (base e) logarithm of x.
`Math.log10(x)`	Returns the base 10 logarithm of x.
`Math.pow(x, y)`	Returns x^y.
`Math.random()`	Returns a random number that is ≥ 0 but < 1.
`Math.sin(x)`	Returns the trigonometric sine of the angle x in radians.

The Class **Math**



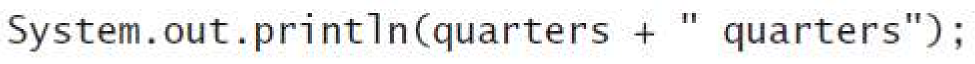
`Math.sin(x)`	Returns the trigonometric sine of the angle *x* in radians.
`Math.sqrt(x)`	Returns the square root of *x*, assuming that $x \geq 0$.
`Math.tan(x)`	Returns the trigonometric tangent of the angle *x* in radians.
`Math.toDegrees(x)`	Returns an angle in degrees equivalent to the angle *x* in radians.
`Math.toRadians(x)`	Returns an angle in radians equivalent to the angle *x* in degrees.

In each of the following methods, the argument and the return value have the same type– either `int`, `long`, `float`, or `double`:

`Math.abs(x)`	Returns the absolute value of *x*.
`Math.max(x, y)`	Returns the larger of *x* and *y*.
`Math.min(x, y)`	Returns the smaller of *x* and *y*.
`Math.round(x)`	Returns the nearest whole number to *x*. If *x* is `float`, returns an `int`; if *x* is `double`, returns a `long`.

Screen Output

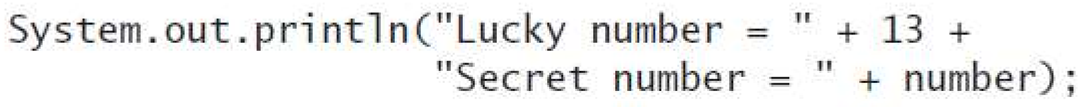
- Statements of the form



```
System.out.println(quarters + " quarters");
```

send output to the screen

- To display more than one thing, simply place a + operator between them

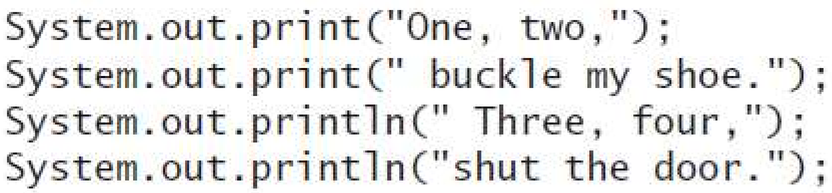


```
System.out.println("Lucky number = " + 13 +
                   "Secret number = " + number);
```

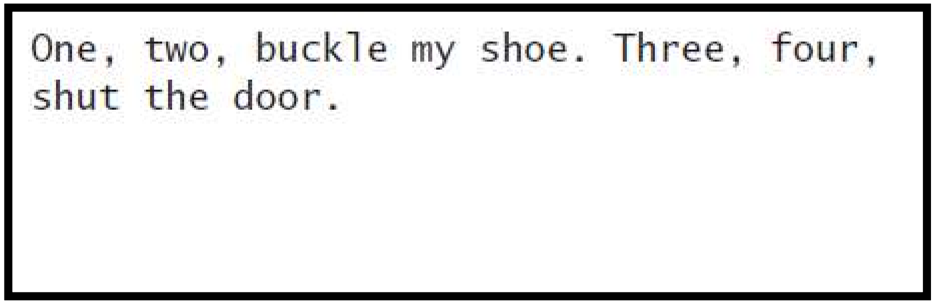
  - the + operator joins, or concatenates, two strings

Screen Output

- Every invocation of **println** ends a line of output
- If you want the output from two or more output statements to appear on a single line, use **print**



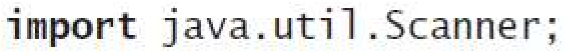
```
System.out.print("One, two,");
System.out.print(" buckle my shoe.");
System.out.println(" Three, four,");
System.out.println("shut the door.");
```



```
One, two, buckle my shoe. Three, four,
shut the door.
```

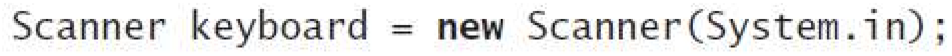
Keyboard Input Using the Class **Scanner**

- Class **Scanner** must be imported
  - Write this line at beginning of program



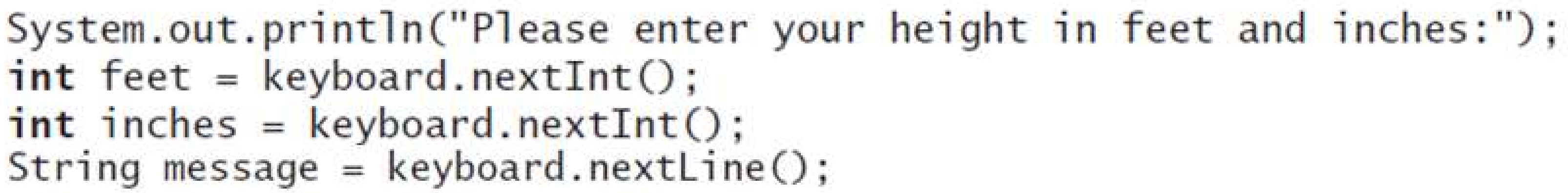
```
import java.util.Scanner;
```

- Must then create a **Scanner** object



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

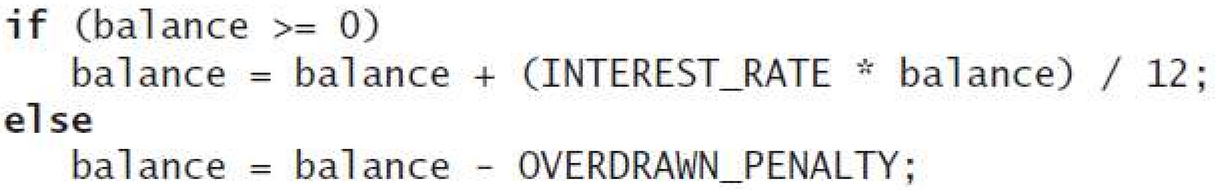
- Read integers, real numbers, strings



```
System.out.println("Please enter your height in feet and inches:");
int feet = keyboard.nextInt();
int inches = keyboard.nextInt();
String message = keyboard.nextLine();
```

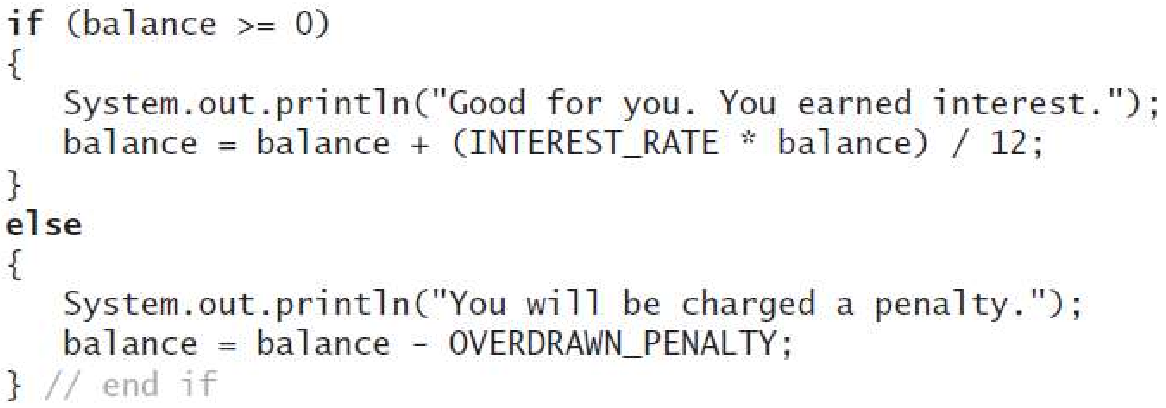
The **if-else** Statement

- Meaning of **if-else** statement, same meaning it would have if read as an English sentence



```
if (balance >= 0)
   balance = balance + (INTEREST_RATE * balance) / 12;
else
   balance = balance - OVERDRAWN_PENALTY;
```

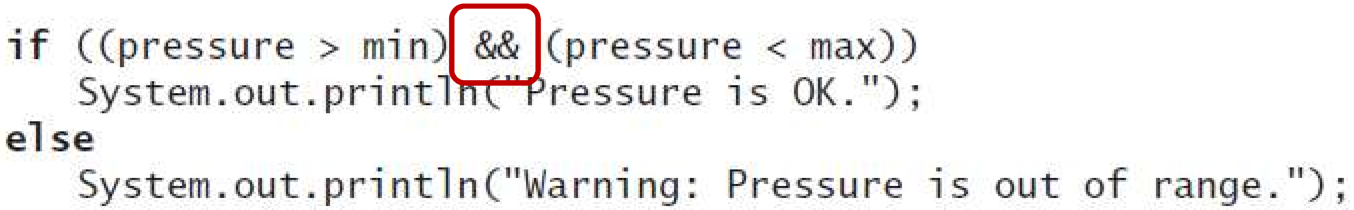
- To include more than one statement, braces



```
if (balance >= 0)
{
   System.out.println("Good for you. You earned interest.");
   balance = balance + (INTEREST_RATE * balance) / 12;
}
else
{
   System.out.println("You will be charged a penalty.");
   balance = balance - OVERDRAWN_PENALTY;
} // end if
```

Logical Operators

- Enables use of boolean expression more complicated than a simple comparison



```
if ((pressure > min) && (pressure < max))
   System.out.println("Pressure is OK.");
else
   System.out.println("Warning: Pressure is out of range.");
```

- Operators
  - Operator && logical **and**
  - Operator || logical **or**
  - Operator ! logical **not**

Logical Operators

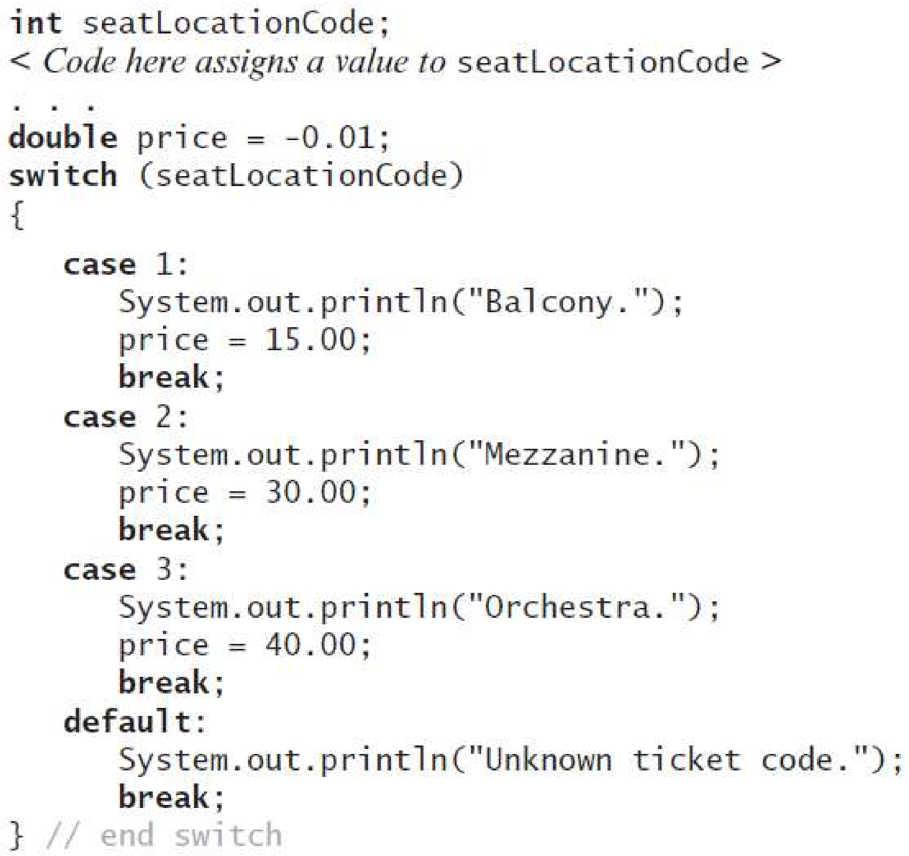
- Precedence of operators
  - The unary operators +, -, !
  - The binary arithmetic operators *, /, %
  - The binary arithmetic operators +, -
  - The comparison operators <, >, <=, >=
  - The comparison operators ==, !=
  - The logical operator &&
  - The logical operator ||
  - Can be overridden with parentheses

The **switch** Statement

- Multiway **if-else** statements can become unwieldy
- If choice is based on value of integer or character expression
  - **switch** statement can make code easier to read
- Begins with word **switch** followed by expression in parentheses
  - Expression must be **int, char, byte, short, String**

The **switch** Statement

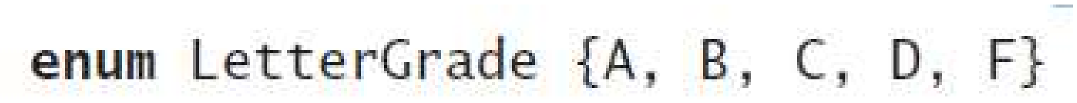
•**switch** statement determines the price of a ticket according to location of seat in theater



```
int seatLocationCode;
< Code here assigns a value to seatLocationCode >
. . .
double price = -0.01;
switch (seatLocationCode)
{
   case 1:
      System.out.println("Balcony.");
      price = 15.00;
      break;
   case 2:
      System.out.println("Mezzanine.");
      price = 30.00;
      break;
   case 3:
      System.out.println("Orchestra.");
      price = 40.00;
      break;
   default:
      System.out.println("Unknown ticket code.");
      break;
} // end switch
```

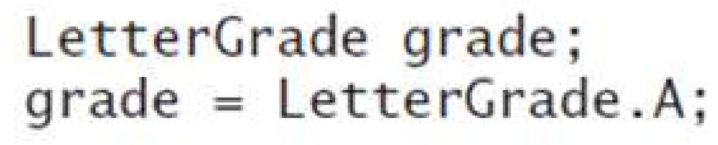
Enumerations

- An enumeration itemizes the values that a variable can have.
- Example: define **LetterGrade** as an enumeration



```
enum LetterGrade {A, B, C, D, F}
```

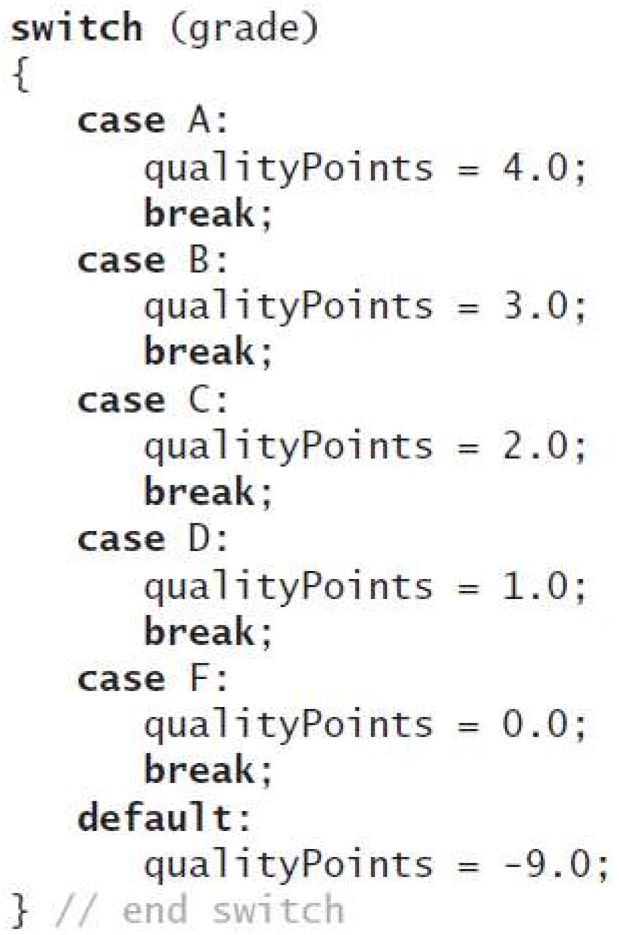
- **LetterGrade** behaves as a class type
  - Values behave as static constants



```
LetterGrade grade;
grade = LetterGrade.A;
```

Enumerations

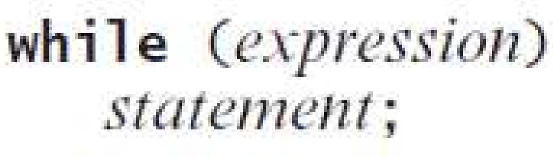
- You can use a **switch** statement with a variable whose data type is an enumeration.



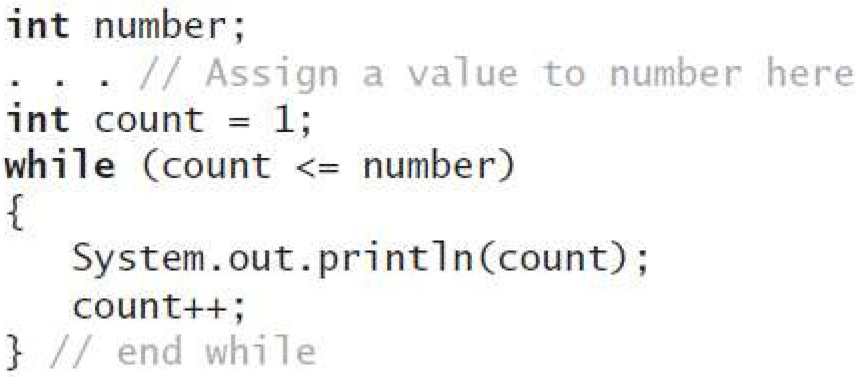
```
switch (grade)
{
   case A:
      qualityPoints = 4.0;
      break;
   case B:
      qualityPoints = 3.0;
      break;
   case C:
      qualityPoints = 2.0;
      break;
   case D:
      qualityPoints = 1.0;
      break;
   case F:
      qualityPoints = 0.0;
      break;
   default:
      qualityPoints = -9.0;
} // end switch
```

The **while** Statement

- General form



```
while (expression)
    statement;
```

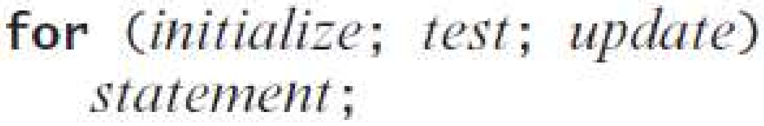


```
int number;
. . . // Assign a value to number here
int count = 1;
while (count <= number)
{
   System.out.println(count);
   count++;
} // end while
```

while statement displays the integers from 1 to a given integer number:

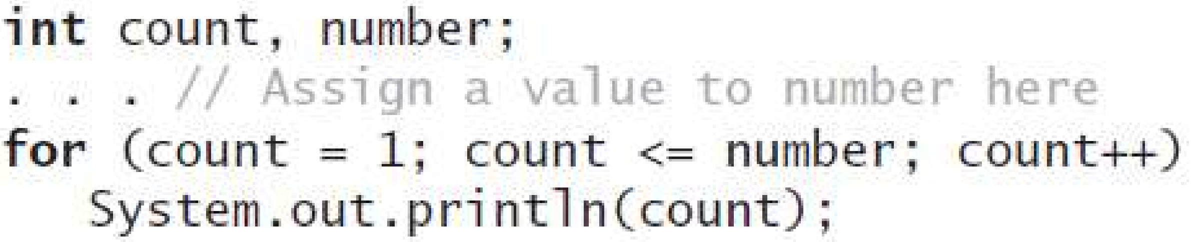
The **for** Statement

- General form



```
for (initialize; test; update)
   statement;
```

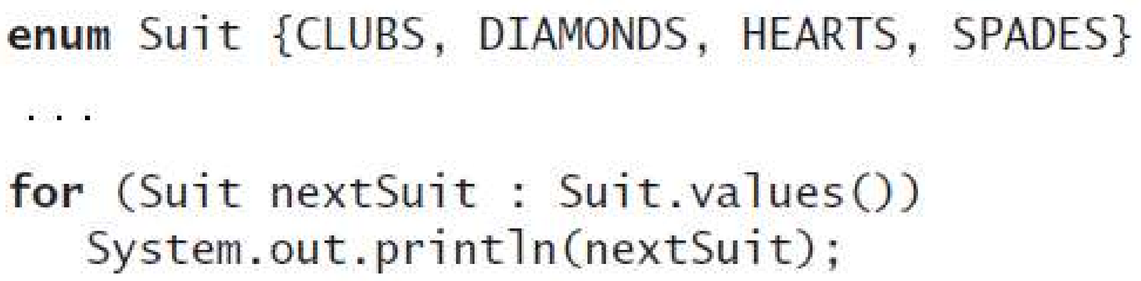
- Same result as while loop shown
  - **for** statement increments for the loop



```
int count, number;
. . . // Assign a value to number here
for (count = 1; count <= number; count++)
   System.out.println(count);
```

The **for** Statement

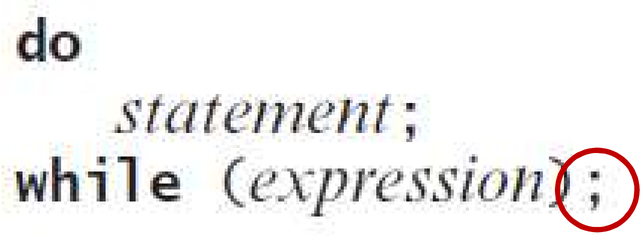
- Using an enumeration with a **for** statement
  - Declare a variable to the left of a colon
  - To right of colon, represent values that variable will have



```
enum Suit {CLUBS, DIAMONDS, HEARTS, SPADES}
 . . .
for (Suit nextSuit : Suit.values())
   System.out.println(nextSuit);
```

The **do-while** Statement

- Similar to the **while** statement
  - But, body of a **do-while** statement always executes at least once
- General form

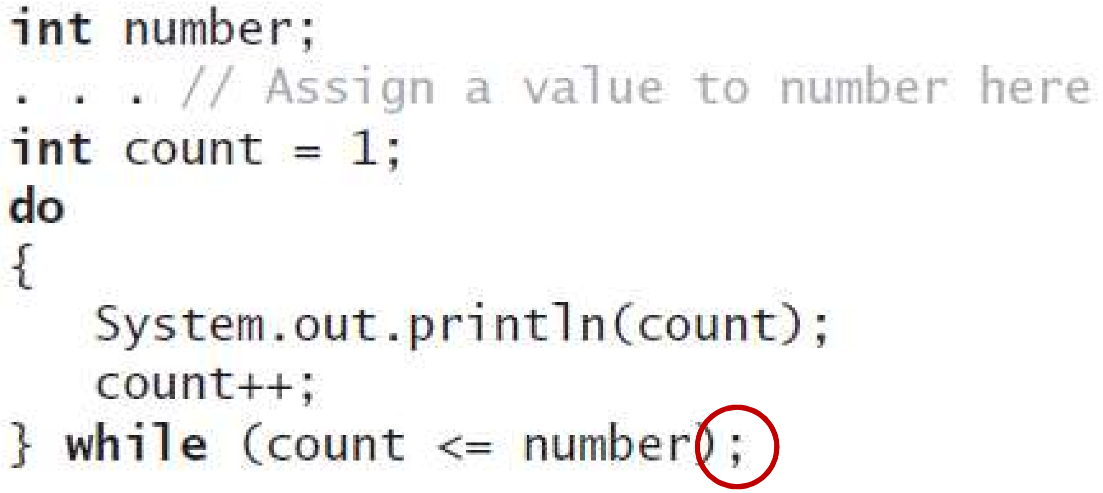


```
do
    statement;
while (expression);
```

  - Be sure to include a semicolon at the end of a **do-while** statement.

The **do-while** Statement

- Be sure to include a semicolon at the end of a **do-while** statement.



```
int number;
. . . // Assign a value to number here
int count = 1;
do
{
   System.out.println(count);
   count++;
} while (count <= number);
```

Additional Loop Information

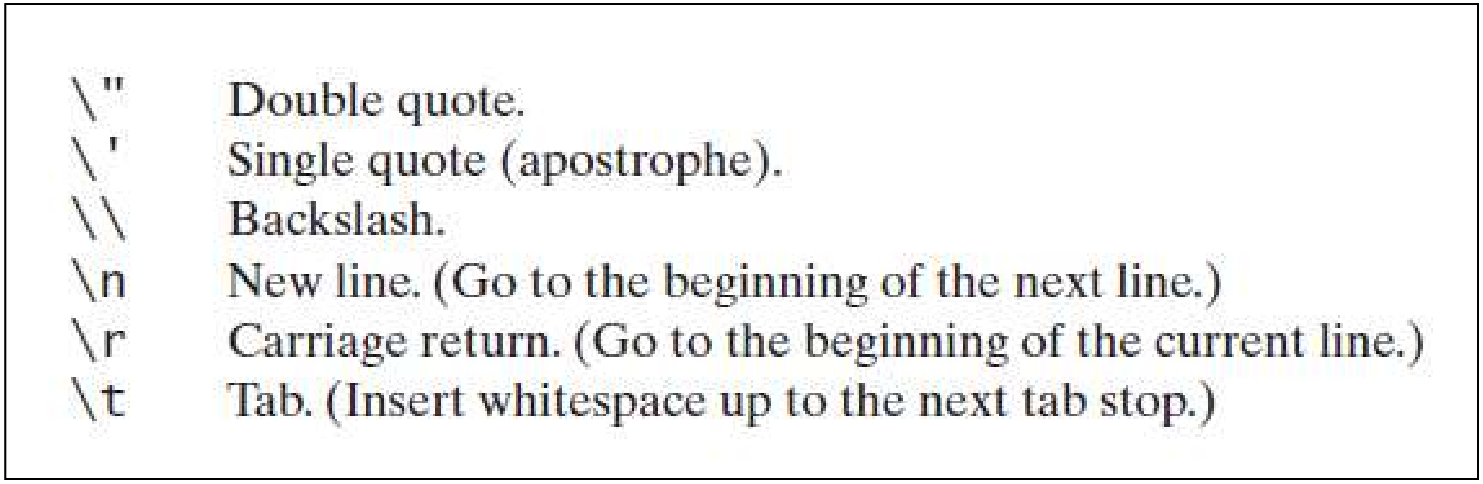
- If loop must run at least one time
  - Use **do-while**
- If loop might not be needed to execute even first time,
  - Use **while-loop**
- **Break** statement can jump out of a loop
- **Continue** statement can jump back to top of loop

The Class **String**

- Part of the package **java.lang** in the Java Class Library
- Use **String** objects to create and process strings of characters.
- Java uses the Unicode character set
  - Codes for ASCII are same in Unicode

The Class **String**

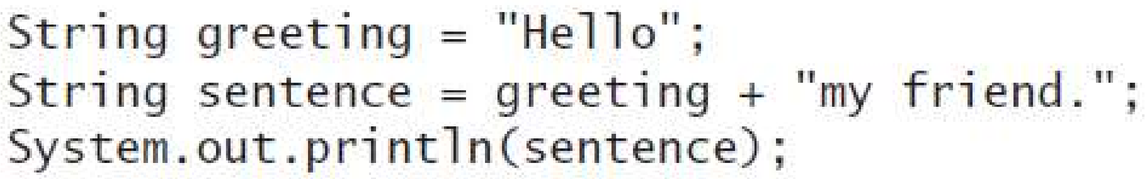
Escape characters



`\"`	Double quote.
`\'`	Single quote (apostrophe).
`\\`	Backslash.
`\n`	New line. (Go to the beginning of the next line.)
`\r`	Carriage return. (Go to the beginning of the current line.)
`\t`	Tab. (Insert whitespace up to the next tab stop.)

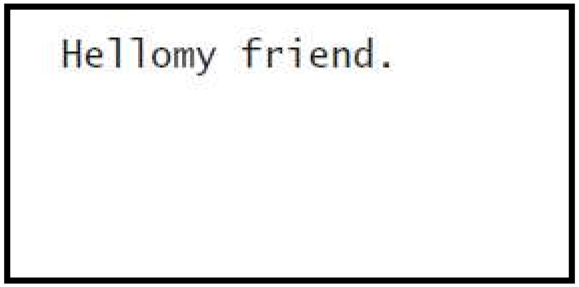
Concatenation of Strings

- Join two strings by using the + operator
  – The concatenation operator for strings



```
String greeting = "Hello";
String sentence = greeting + "my friend.";
System.out.println(sentence);
```

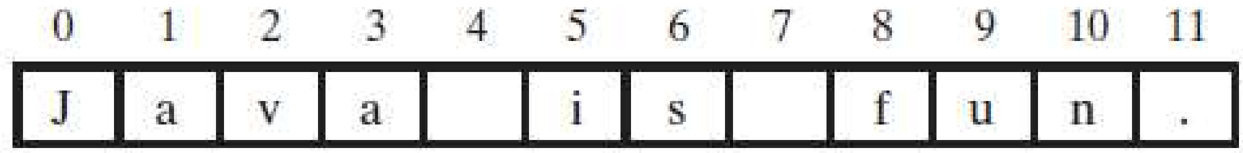
- Result displayed on screen is



```
Hellomy friend.
```

String Methods

- **String** object has methods as well as a value
  - Use these methods to manipulate string values
- **length** gets number of characters in a string
- Use the **concat** instead of the + operator



0	1	2	3	4	5	6	7	8	9	10	11
J	a	v	a		i	s		f	u	n	.

String Methods

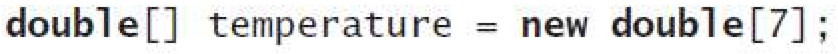
- **charAt** returns the character at the index given
  - If index negative or too large, causes error
- **indexOf** tests whether string contains given substring
  - If it does, returns index at which substring begins
- **toLowerCase** replaces uppercase letters with their lowercase counterparts of argument

String Methods

- **trim** trims off leading, trailing white space
- Use method **compareTo** to compare two strings – lexicographically
  s1.compareTo (s2) returns
  - negative integer if s1 < s2
  - positive integer if s1 > s2
  - zero if s1 = s2

Arrays

- A special kind of object that stores a finite collection of items having the same data type

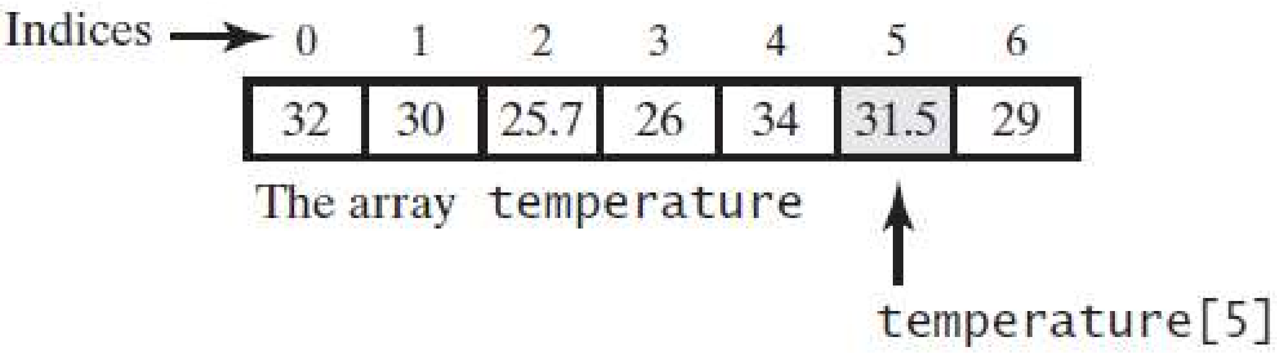


```
double[] temperature = new double[7];
```

  - Left side of assignment operator declares **temperature** an array whose contents are of type double.
  - Right side uses **new** operator to request seven memory locations for array
  - Number in brackets – the index, integer value

Arrays

- An array of seven temperatures

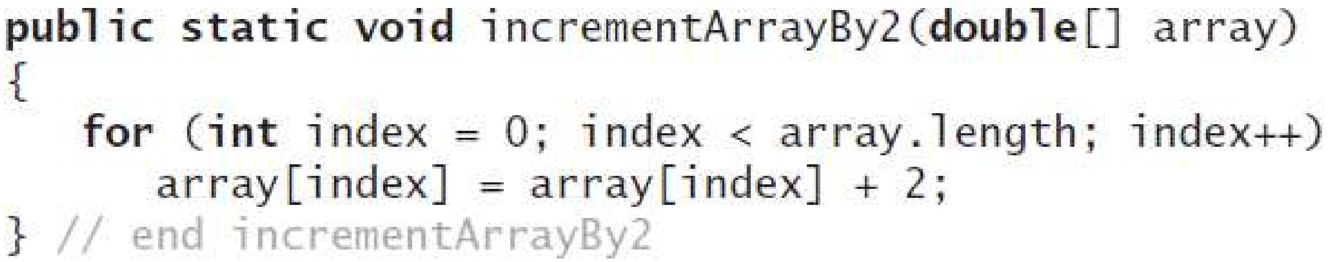


0	1	2	3	4	5	6
32	30	25.7	26	34	31.5	29

- Note: array is full, each location has a value
- Arrays are hot always full – must distinguish between length and number of items currently stored

Array Parameters and Returned Values

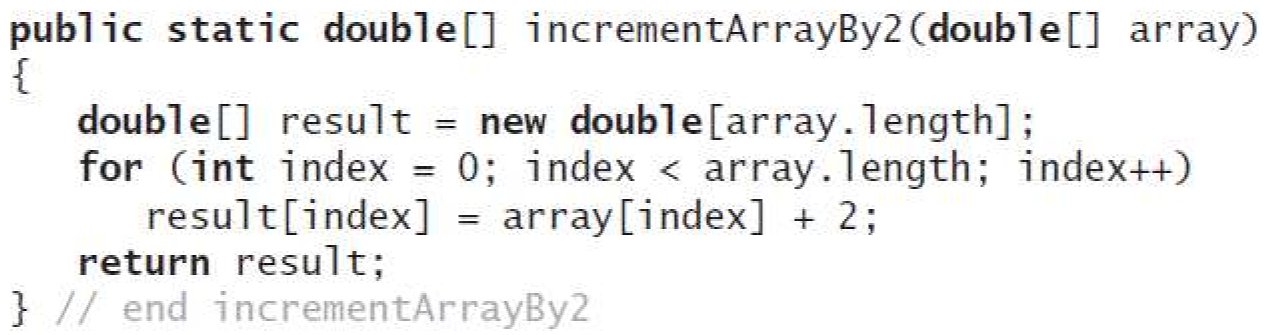
- You can pass indexed variable as argument to a method
  - Anyplace you can pass ordinary variable of array’s entry type.
- An entire array can also be a single argument to a method



```
public static void incrementArrayBy2(double[] array)
{
   for (int index = 0; index < array.length; index++)
      array[index] = array[index] + 2;
} // end incrementArrayBy2
```

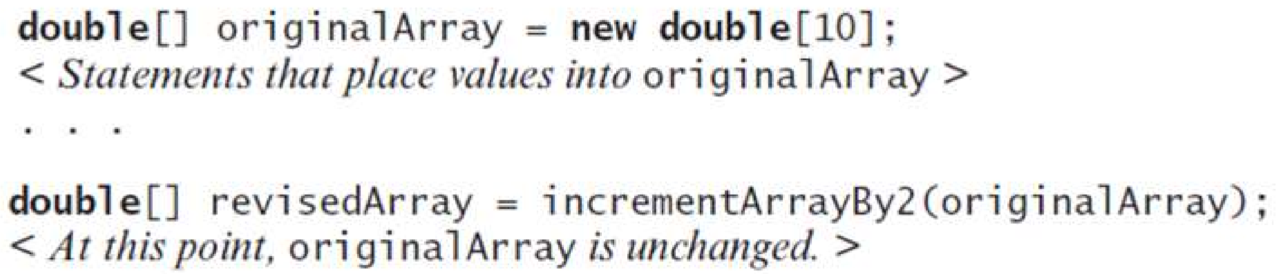
Array Parameters and Returned Values

- A method can return an array



```
public static double[] incrementArrayBy2(double[] array)
{
   double[] result = new double[array.length];
   for (int index = 0; index < array.length; index++)
      result[index] = array[index] + 2;
   return result;
} // end incrementArrayBy2
```

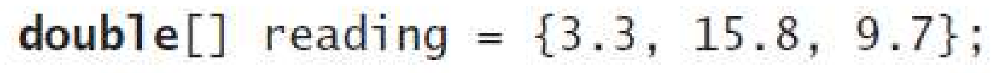
  - –Call of this method …



```
double[] originalArray = new double[10];
< Statements that place values into originalArray >
. . .
double[] revisedArray = incrementArrayBy2(originalArray);
< At this point, originalArray is unchanged. >
```

Initializing Arrays

- Provide initial values for the elements in an array when you declare it

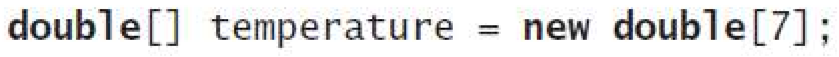


```
double[] reading = {3.3, 15.8, 9.7};
```

- You do not explicitly state array’s length.
  - Length is minimum number of locations that will hold given values

Array Index Out of Bounds

- Consider this array

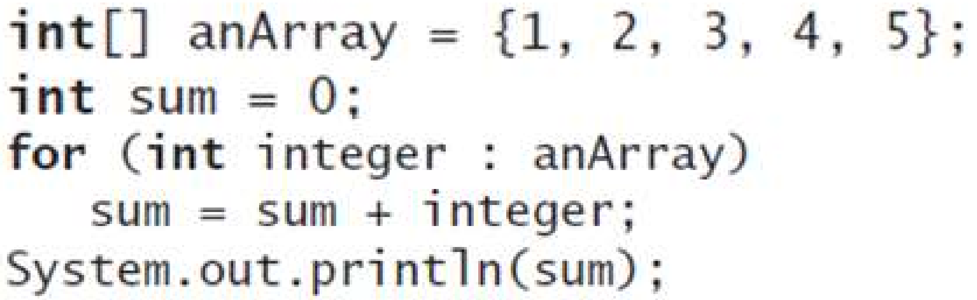


```
double[] temperature = new double[7];
```

  - If index is negative or greater than 6, it is said to be “out of bounds”
- If index is an expression and out of bounds
  - Causes an **IndexOutOfBoundsException**

Arrays and the For-Each Loop

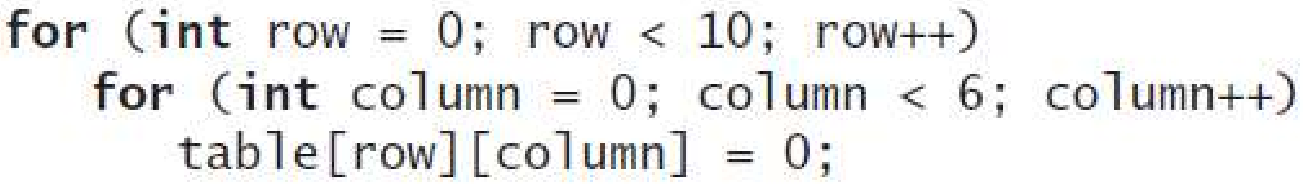
- Can use **for-each** loop to process all the values in an array



```
int[] anArray = {1, 2, 3, 4, 5};
int sum = 0;
for (int integer : anArray)
   sum = sum + integer;
System.out.println(sum);
```

Multidimensional Arrays

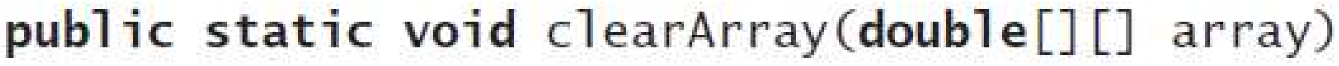
- A loop that will set all the values of **table** to zero



```
for (int row = 0; row < 10; row++)
   for (int column = 0; column < 6; column++)
      table[row][column] = 0;
```

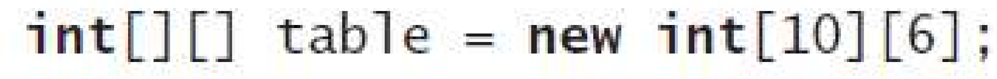
- Multidimensional array can be parameter of a method

  – Above loop could be placed in a method of this name



```
public static void clearArray(double[][] array)
```

Multidimensional Arrays

- Java implements multidimensional arrays as one-dimensional arrays
  - Given 

```
int[][] table = new int[10][6];
```

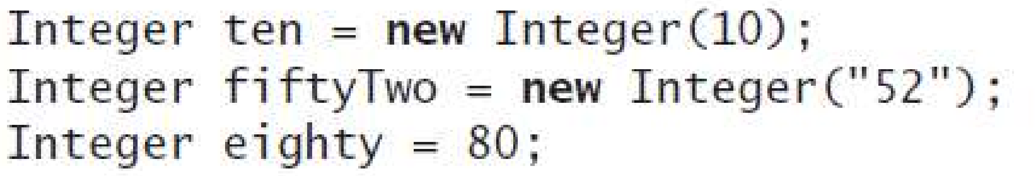
- Array **table** is in fact a one-dimensional array of length 10, and its entry type is **int[]**
- In other words, a multidimensional array is an array of arrays

Wrapper Classes

- An argument to a method and the assignment operator = behave differently for primitive types and class types
- To make things uniform, Java provides a wrapper class for each of primitive types
  - Enables conversion of a value of primitive type to object of corresponding class type.

Wrapper Classes

- Example: we want to convert an **int** value, such as 10, to an object of type **Integer**
  - Can be done in one of three ways

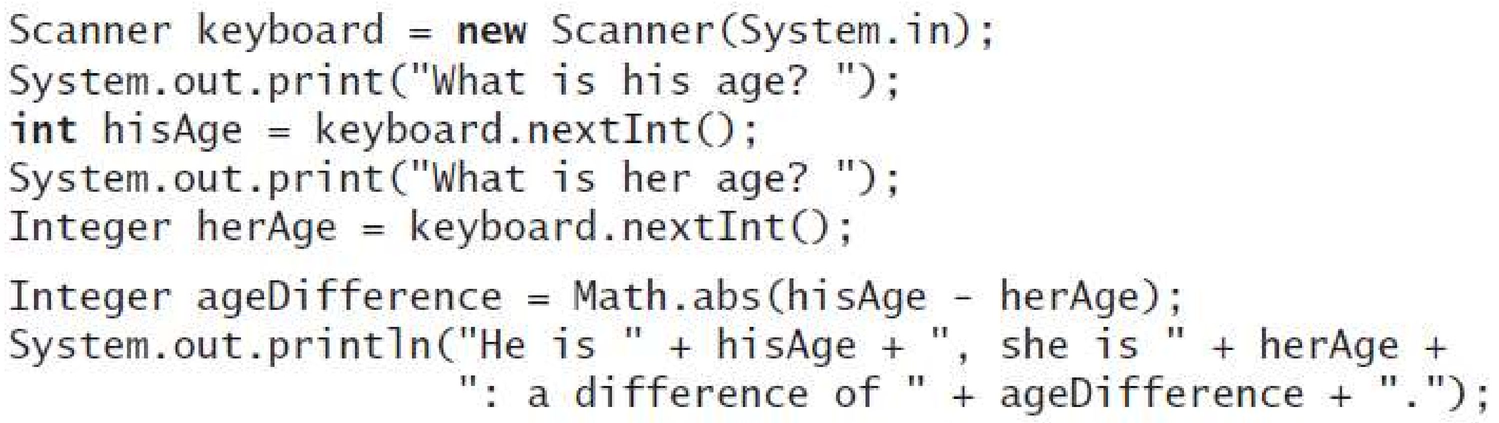


```
Integer ten = new Integer(10);
Integer fiftyTwo = new Integer("52");
Integer eighty = 80;
```

- Now use methods **equals** and **compareTo** for comparisons
  - Do not use **==** for comparisons or **=** for assignments as with primitives

Wrapper Classes

- You can use same operators that you use for arithmetic with primitives

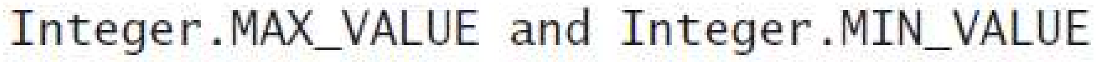


```
Scanner keyboard = new Scanner(System.in);
System.out.print("What is his age? ");
int hisAge = keyboard.nextInt();
System.out.print("What is her age? ");
Integer herAge = keyboard.nextInt();

Integer ageDifference = Math.abs(hisAge - herAge);
System.out.println("He is " + hisAge + ", she is " + herAge +
                   ": a difference of " + ageDifference + ".");
```

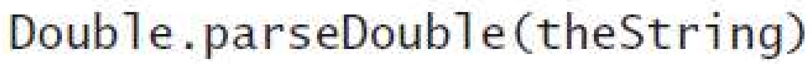
Wrapper Classes

- Wrapper classes contain useful static constants
  - The largest and smallest values of type **int** are



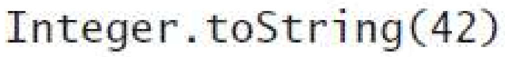
```
Integer.MAX_VALUE and Integer.MIN_VALUE
```

  - Methods that can be used to convert a string to the corresponding numerical type



```
Double.parseDouble(theString)
```

  - Or back the other direction



```
Integer.toString(42)
```

Wrapper Classes

- **Character** is the wrapper class for the primitive type **char**
- Some of the methods include
  - **toLowerCase, toUpperCase**
  - **isLowerCase, isUpperCase**
  - **isLetter, isDigit, isWhitespace**