Appendices

Java Basics

- Compiling and Executing
- Java Syntax and Types
- Classes, instantiation, constructors, overloading
- References, Strings
- Garbage Collection
- Arrays
- Dynamic Array Types, Vectors, ArrayLists
- Parameter Passing
- Differences between Java and C++

SAMPLE PROGRAM:

- Save the following code in file "HelloWorld.java":

```
import java.lang.*;
public class HelloWorld {
     public static void main ( String [ ] args ) {
          Greeter newGreeter = new Greeter();
          newGreeter.sayHello();
          }
     }
```

- Save the following code in file "Greeter.java"

```
public class Greeter{
     public void sayHello()    {
          System.out.println ("Hello World!") ;
     }
}
```

- Compile using the command

```
javac   HelloWorld.java
```

(implicitly also compiles Greeter.java)

- Execute HelloWorld using the command

```
java   HelloWorld
```

COMPILING:

```
% javac   MyProg.java          // tells the Java Compiler to compile the Java
                               //  source code in the file named "MyProg.java";
                               //  file "MyProg.java " must contain a 'class'
                               //  whose name is "MyProg";
                               //  produces an output "bytecode" file named
                               //  "MyProg.class"
```

EXECUTING:

```
% java   MyProg                // runs the JVM ("Java Virtual Machine"),  tells
                               // it to "interpret"  (execute)
                               //  the byte code in file "MyProg.class"
```

- File and Class names are case-sensitive (even in non-sensitive OS environments such as Windows)
- Source program file names *must* end in ".java"

ENVIRONMENT:

- Path to Java tools ("java", "javac") must be added to the "PATH" variable; e.g. in Windows command line:

  set PATH=C:\Program Files\Java\jdk1.8.0_20\bin;%PATH%

- Path to the current directory (indicated by a single period) must be included in the "CLASSPATH" variable; e.g. in Windows command line:

  set CLASSPATH=.;%CLASSPATH%

- Path to the Java home directory must be defined as the "JAVA_HOME" variable; e.g. in Windows command line:

  set JAVA_HOME= C:\Program Files\Java\jdk1.8.0_20

Note that "set" commands mentioned above set the variables temporarily (will only be valid for processes that will be launched from the command window that we have typed the "set" command). **To set them permanently in Windows, go to "Control Panel -> System -> Advance System Settings -> Environment Variables (add it to "System Variables")".**

Java Built-in Primitives (8 kinds):

INTEGER types:

- **byte** (-128 .. +127)
- **short** (2 bytes: -32768 .. +32767)
- **int** (4 bytes: -2G .. +2G)
- **long** (8 bytes: $\pm 2^{63}$)

REAL types:

- **float** (4 bytes, IEEE std., values denoted like "1.0F")
- **double** (8 bytes, IEEE std., values denoted like "1.0")

Additional types:

- **boolean** (*true* or *false)*
- **char** (16-bit "Unicode")

<u>Variable Declarations (primitives):</u>

```
int  a ;

long  j = 4271843569L ;

double  x = 1.378 ;

char  c1 = ‘a’,   c2 = ‘z’ ;


int  i = 2 ;
int  k = i + 3 ;
```

Strong Typing:

```
float  x = 1.0 ;     // fails!   Must be 1.0F   (use double)


int j = 1 + ‘z’ ;           // fails!  Cannot mix types (unless “casting” is used)
```

<u>MODIFIERS:</u>

- Used to specify access and/or usage of classes, fields, and/or methods

- Visibility Modifiers

  - **`public`** // "world accessible"
  - **`private`** // "only accessible by methods in this class"
  - **`protected`** // "accessible by all classes in this 'package',
    // and all subclasses in *any* package"
  - <default> // "accessible by any class in this package"

- Additional Modifiers

  - **`static`** //"one for the whole class"
  - **`final`** //"restricted use" – e.g. variable cannot be changed
  - others to be seen later…

```
/** This is a sample class whose purpose is simply to show the form of several Java constructs. The class
  * provides a method which computes and prints the sum of all Odd integers between 1 and a  fixed Max
  *which do not lie inside a specified "cutoff range", and also are either divisible by 3 but  are not certain
  * "special rejected" numbers, or else are divisible by 5.   It does the calculation three times, expanding the
  * cutoff range each time.  It is assumed the Calculator is instantiated, and findSum is invoked, elsewhere.
  */
public class Calculator {
        public void findSum ()  {
                final int MAX = 100 ;
                int loopCount = 0 ;
                int lowerCutoff = 50;
                int upperCutoff = lowerCutoff + 25;
                int rangeExpansion = 10 ;

                while (loopCount < 3) {
                        int sum = 0 ;
                        for (int i=1; i<=MAX; i++) {
                                //check for odd number outside cutoff range
                                if ( (i%2 != 0) && ((i<lowerCutoff) || (i>upperCutoff)) ) {
                                        //found an acceptable odd number; check if divisible by 3
                                        if ((i%3 == 0))  {
                                                switch (i)   {  //divisible by 3; check for special reject numbers
                                                     case 15: {
                                                             System.out.println ("Found and rejected 15");
                                                             break;
                                                     }
                                                     case 21:
                                                     case 27: {
                                                             System.out.println ("Found and rejected " + i);
                                                             break;
                                                     }
                                                     default: { sum = sum + i ; }
                                                }
                                        }  else {
                                                //not divisible by 3; check if multiple of 5
                                                if (i%5 == 0)
                                                     sum = sum + i ;
                                        }
                                }
                        }
                        System.out.println ("Loop " + loopCount + ":  sum = " + sum );
                        lowerCutoff -= rangeExpansion; //increase the cutoff range
                        upperCutoff += rangeExpansion;
                        loopCount++;
                }
        }
}
```

CLASSES:

- Nearly every data item in a Java program is an **OBJECT**
    - (primitives are the exception)
- An *object* is an **INSTANCE** of a **CLASS**
    - Programmer-defined class, or
    - Class from a predefined library
- All code in a Java program is inside some class – even the *main* program
- Classes contain **FIELDS** and **METHODS** (also called *procedures* or *functions*)

```
public class  BankAccount {
    private double currentBalance ;

    private String ownerName = "Rufus" ;

    public int branchID = 405 ;


    public double getBalance() {
        return currentBalance ;
    }

    public void deposit (float amount){
        currentBalance += amount ;
    }
}
```

INSTANTIATION:

- Primitives (int, char, boolean, etc.) are not objects (in the OO/Java sense)
    - allocated as “local variables” on the stack
- Code in a class (e.g. the class containing the main program) can create objects by **INSTANTIATION**
    - Objects are allocated on the Dynamic Heap
- Example instantiations:

```
//assume the following user-defined class:
  public class Ball {
       private int xCenter, yCenter, radius;
       private Color ballColor ;
       //other fields here . . .
       //method declarations here . . .
  }


//the following statements create INSTANCES of the Ball
class:

  Ball  myBall  = new Ball();

  Ball  yourBall = new Ball();

//the following statement creates an (initialized) INSTANCE
//of the predefined Java class String:

  String myName = new String (“Rufus T. Whizbang”);
```

CONSTRUCTORS:

- **Instantiation** is done using the **new** operator to invoke a "**CONSTRUCTOR**"
  - ➢ **No "implicit instantiation" like in C++**

    `Ball myBall ;` // **creates** a Ball object in C++, but not in Java!

- Constructors always have exactly the same name (including case) as the CLASS
- Task of a constructor: **Create** and **Initialize** an object (an instance of the class)
- Programmer can define multiple constructors with different parameters (arguments)
- If the programmer provides NO constructors for a class, Java automatically provides a 'default' constructor with no parameters ("default no-arg constructor")
- NOTE: no "destructor" in Java [C++ "~" ; C "**free**" ; Pascal "**dispose**"]
  - ➢ **Objects are automatically "freed" when no longer accessible – handled via "garbage collection"**

CONSTRUCTOR EXAMPLES:

```
//assume the following user-defined class:
  import java.awt.Color;
  public class Ball {
        private int xCenter, yCenter, radius;
        private Color ballColor ;

        // (programmer-provided) no-arg constructor
        public Ball () {
              xCenter = yCenter = 0;
              radius = 1;
              ballColor = Color.red;
        }

        // constructor allowing specification of Color
        public Ball(Color theColor) {
              ballColor = theColor ;
              xCenter = yCenter = 0;
              radius = 1;
        }

        //methods (functions) provided by "Ball" objects
        public int getDiameter() {
              int diameter = 2 * radius ;
              return diameter ;
        }
  }


// - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
//then the following would be typical instantiations appearing
//  in code in some program (class):
  . . .
  Ball myBall = new Ball();                //a red ball of radius 1 at (0,0)

  Ball yourBall = new Ball(Color.blue);    //a blue ball, radius=1 at (0,0);
  . . .


  //invocations of methods in different objects (instances):
  int myDiameter = myBall.getDiameter();
  int yourDiameter = yourBall.getDiameter();
```

Overloading Constructors and Methods:

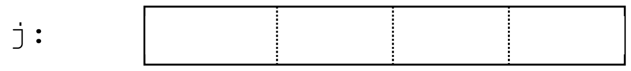
(example from Jia: OO Software Development, A-W, 2000)

```
/** This class gives a representation of a point,
  * showing examples of overloading.
  */
public class Point {
      private double x,y ;    //the coordinates of the point
      //overloaded constructors:
      public Point ()   {
            x = 0.0 ;
            y = 0.0 ;
      }
      public Point (double xVal, double yVal)   {
            x = xVal ;
            y = yVal ;
      }
      //overloaded methods:
      /** Returns the distance between this point and the other point */
      public double distance (Point otherPoint) {
            double dx = x - otherPoint.x ;
            double dy = y - otherPoint.y ;
            return Math.sqrt (dx*dx + dy*dy) ;
      }
      /** Returns the distance between this point and a location */
      public double distance (double xVal, double yVal)     {
            double dx = x - xVal ;
            double dy = y - yVal ;
            return Math.sqrt (dx*dx + dy*dy) ;
      }
      /** Returns the distance between this point and the origin */
      public double distance ()     {
            return Math.sqrt (x*x + y*y) ;
      }
}
```

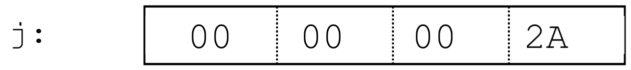
REFERENCES:

- A variable of a primitive type holds a data value of that type:

```
int  j ;        //declaration allocates space for j:
```

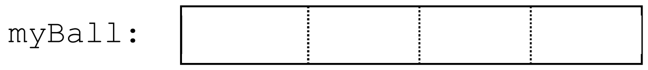


```
j = 42 ;                  //assignment stores a value in the space:
```



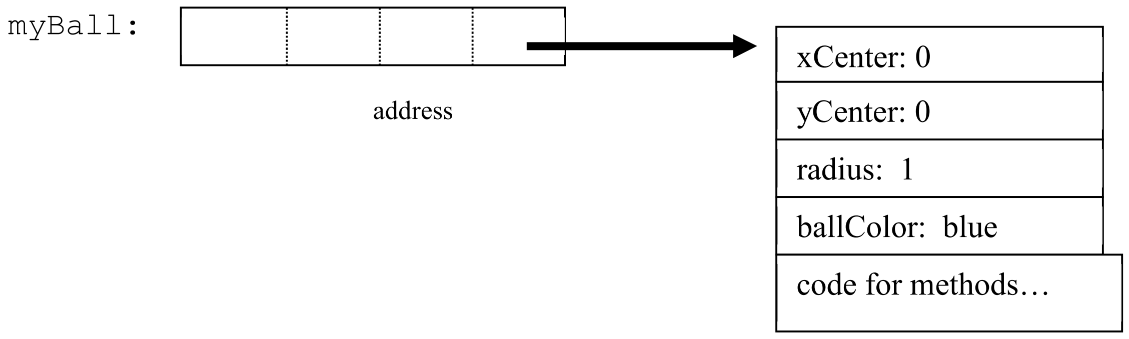
- A variable representing an object holds the address of the object, called a *reference*:

```
Ball myBall ;  //declaration allocates space for a pointer:
```



- Construction of an object allocates space on the Heap for the object, and sets the reference to point to the object:

```
myBall = new Ball(Color.blue);     // create the object:
```



<u>**REFERENCES, cont. :**</u>

- All object values (fields and methods) are accessed via a reference:

```
Ball myBall = new Ball(Color.blue);

Color myColor = myBall.ballColor;  //access object color

int diameter = myBall.getDiameter();  //invoke object
method
```

- A "reference" is essentially a <u>pointer</u> that doesn't have to be "dereferenced":

  Pascal-like dereference :

```
     myBall : pointer to Ball ;
     myColor :=  myBall^ballColor;
```

  C-like dereference :

```
     Ball * myBall ;
     myColor =  myBall -> ballColor;
```

- Java <u>***only***</u> has references – no other way to access objects

- Java does not allow "pointer (reference) arithmetic"

REFERENCES vs. PRIMITIVES:

- Consider the following code which uses primitives:

```
...
int a = 42;
int b = a;
System.out.println (a);       //prints 42
b = 3;
System.out.println (a);       //still prints 42
```

- Now consider the following analogous code using objects (hence references):

```
     //assume we have the following class definition:
     class Point {
          int x, y;
          public Point (int xVal, int yVal) {
               x = xVal;   y = yVal;
          }
     }

...

Point a = new Point (6,4);
Point b = a;
System.out.println (a.x);          // prints 6
b.x = 13;
System.out.println (a.x);          // prints 13 !!
```

TESTING REFERENCES:

- Consider the following code (assume Class Point as before):

```
...
Point p1 = new Point(0,0);
Point p2 = new Point(0,0);    //another point with same
values
if (p1 == p2) {
     System.out.println ("The points are equal");
}
```

  This will ***not*** print the message; "==" tests if the items (references) are equal

- The following WILL print the message, since the references are equal:

```
...
Point p1 = new Point(0,0);
Point p2 = p1;
if (p1 == p2) {
     System.out.println ("The points are equal");
}
```

- To check if object *contents* are equal, the object must have an "equals()" method:

```
...
if (p1.equals(p2)) {
     System.out.println ("The points are equal");
}
```

- Many Java-defined classes *(e.g. String)* do have "equals()" methods – but not all.

STRING REFERENCES:

- Using and testing **Strings** sometimes causes confusion, but the same rules apply:

```
...
String s1 = new String("Ed");
String s2 = new String("Ed"); //a different String object
if (s1 == s2) {
     System.out.println ("The strings are equal");
}
```

  This will ***not*** print the message

- The following examples all WILL print the message:

```
...
if (s1.equals(s2)) {  ...  }

if (s1.equals("Ed"))  {  ...  }

if (s1.equalsIgnoreCase("ed"))  {  ...  }
```

  Class String defines both "equals()" and "equals**I**gnoreCase()"

- A common mistake:

```
...
if (s1 == "Ed") {
     System.out.println ("The string contains 'Ed' ");
}
```

  This will ***not*** print the message

- Correct approach:

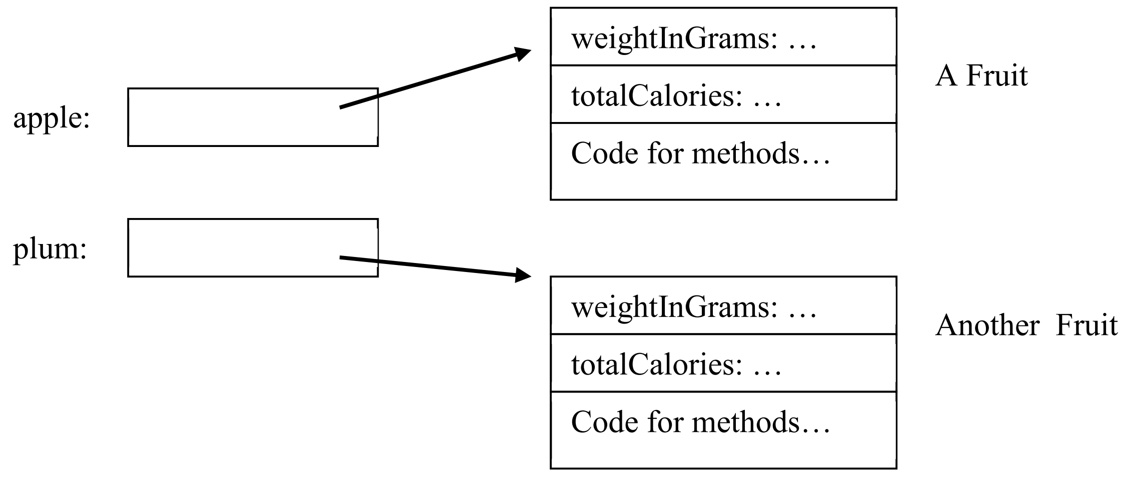
```
if (s1.equals("Ed"))  { ... }
```

<u>GARBAGE COLLECTION:</u>

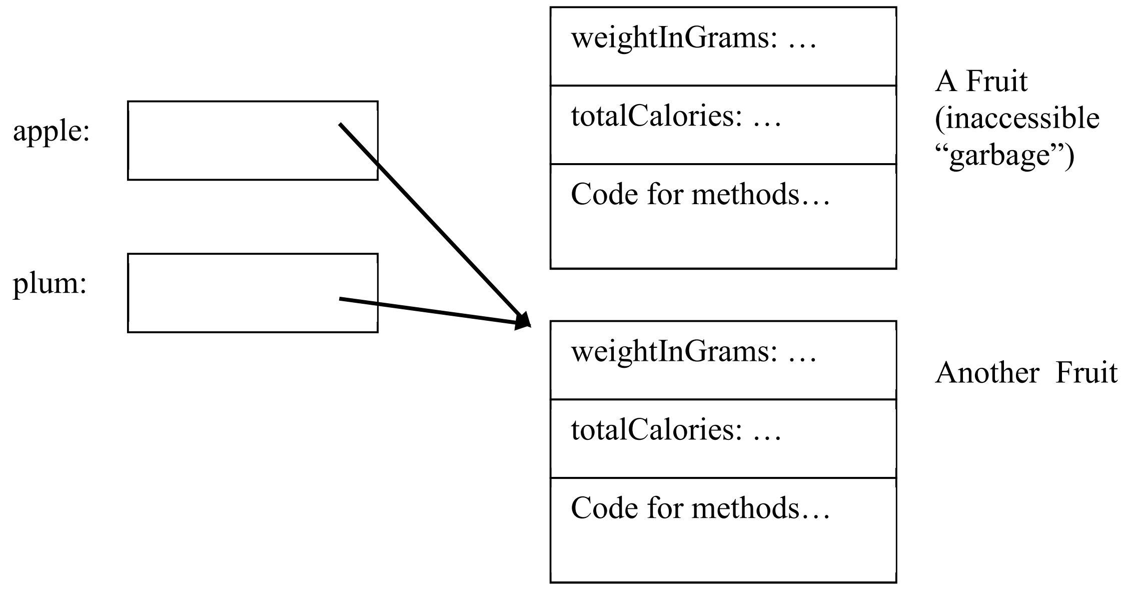
- Important characteristic: assignment does not copy <u>objects</u>; it copies <u>references</u>:

```
Fruit apple = new Fruit();
Fruit plum = new Fruit();
. . .
apple = plum ;      //reference to apple replaced;
                    // now points to same object as "plum"
```

Before assignment:



After assignment:



<u>ARRAYS:</u>

<u>Declaration:</u>

```
int  [ ]  vals ;                    // or:  int  vals  [ ] ;
int  [ ]  scores1, scores2 ;       // two arrays of integers
char  [ ]  grades, letters ;       // two arrays of chars
Point [ ] myPoints ;               // array of objects
```

<u>Characteristics:</u>

- ***<u>Arrays are objects</u>*** (regardless of the type of data in them – primitive or object)
- Like all objects, arrays must be *instantiated:*

```
int  [ ]  vals  = new int [size] ; //size = # elements

char grades [ ] = new char [5] ;   // five elements

Point [ ] myPoints = new Point [myScreen.getSize()] ;

String [ ] words = new String [25]; // holds 25 String refs
```

- Arrays are allocated *<u>on the Heap</u>* (like all objects)
- <u>Size</u> and <u>element-type</u> are fixed at compile time (see “Vector” and “ArrayList” classes)

ARRAYS, cont:

Common Declaration mistakes:

```
int [size] vals ;        //ILLEGAL - need 'new'

int vals [size];         //ALSO ILLEGAL
```

Indexing:

- Even though they are *objects*, special syntax allows "normal" indexing:

```
vals [5] = 99 ;                     // store primitives

grades [3] = 'D' ;

myPoints [i] = new Point (4,3) ;    // store an object
```

- Indexing range is **0 .. size-1** -- like C
- Runtime range checking is enforced

Arrays as Objects:

- Array names are *references* – "pointers to the array object"
- Like all objects, arrays have *FIELDS* and *METHODS*

```
vals.length    // field (not method) giving array size;
               // length is "final" - cannot be changed
```

INITIALIZERS:

```
int [ ] vals  = { 1, 3, 17, 99 } ;
char [ ] letterGrades = { 'A', 'B', 'C', 'D', 'F' } ;
```

- Implicit instantiation (no *new* needed)
- Size of list determines array size
- Only allowed in a declaration – not runtime assignable:

```
int [ ] vals  = { 1, 3 } ;    // OK
. . .
vals = { 1, 4, 7 } ;          // ILLEGAL (in any form)
```

ARRAYS and REFERENCES:

Potential Confusion:

```
        int [ ] myInts  = new int [10] ;

        System.out.println (myInts[4]) ;        // prints '0'

        . . .

        myInts[4] = 1776 ;

        System.out.println (myInts[4]) ;        // prints '1776'
```

but:

```
        Ball [ ] myCollection  = new Ball [10] ;

        System.out.println (myCollection[4]) ;      // runtime
    error!

        . . .

        myCollection [4] = new Ball (Color.blue) ;

        System.out.println (myCollection[4]) ;

             // prints string rep of Ball (if rep exists; else error)
```

- Primitives are initialized to data; Object references are initialized to **`null`**

ARRAYS and REFERENCES, cont:

Another easy “reference” ‘slip-up’:

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

int [ ] b  = { 1, 2, 3 } ;  //identical data

. . .

if ( a = = b ) {

     System.out.println (“arrays ‘a’ and ‘b’ are equal);

}
```

- The above code does NOT print the message….

- Solution: “**java.util.Arrays**” [JDK 1.2 (and up)]
  - Contains methods which operate on arrays
  - Method **equals()** does an element-by-element comparison:

```
if ( Arrays.equals(a,b) ) {

     System.out.println ("a and b are equal") ;

}
```

  - Uses “==” for testing primitives
  - Uses (expects) a **.equals()** method to be defined for objects in arrays

ARRAYS OF ARRAYS:

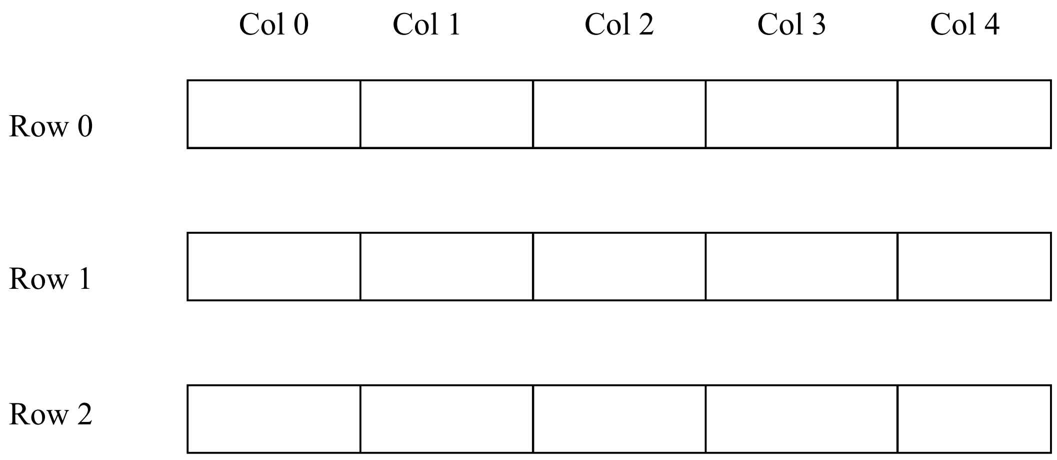
Declaration :

```
int  [ ]  [ ] intTable ;      //"2D array" of ints

Point [ ] [ ] pointTable ;    //"2D array" of Points
```

Instantiation :

```
intTable = new int [3] [5] ;  //could combine with decl.

pointTable = new Point [3] [5] ;
```

Result:



Accessing:

```
intTable [0] [2] = 5 ;        //assigns a primitive
pointTable [0] [2] = new Point (17,-6) ; //assigns an object

for (int i=0; i<3; i++) {
     for (int j=0; j<5; j++) {
          intTable [i] [j] =  i * j  ;
          pointTable [i] [j] = new Point (i, j) ;
     }
}
```

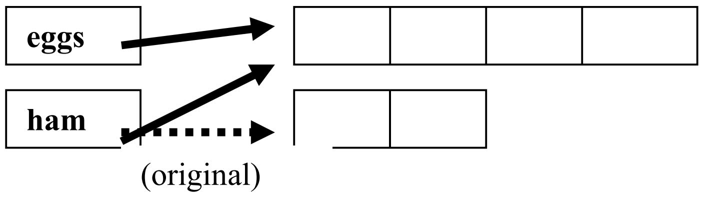
ARRAY ASSIGNMENT:

Arrays can be assigned to another array *if* they have:

- Same element type
- Same number of dimensions:

```
int [ ] eggs  = { 1, 2, 3, 4 } ;

int [ ] ham   = { 1, 2 } ;

. . .

ham = eggs ;   //legal - same element type (int) and dimension

ham [3] = 0 ;  //legal – ham now has 4 elements!
```

- This works because *references* are what is being "assigned" (copied):

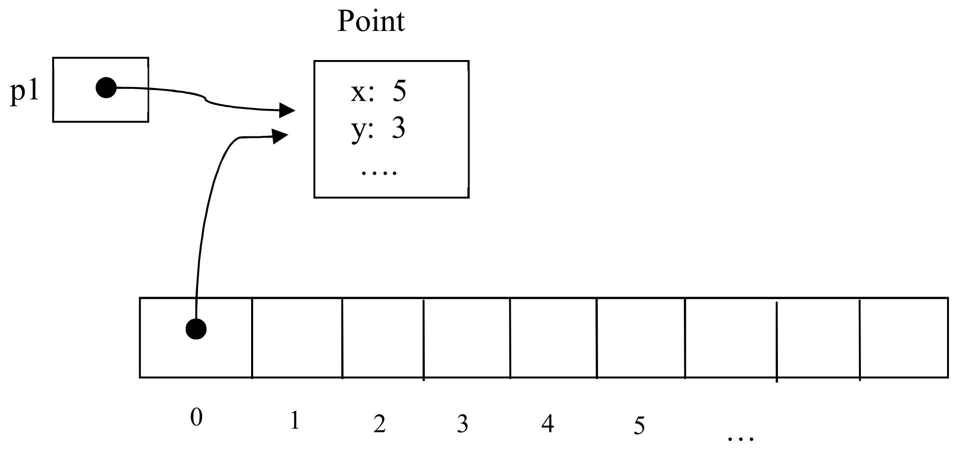


- Method ***System.arraycopy( )*** can be used to perform a real "copy"

Vectors and ArrayLists :

- *Dynamic* *"arrays"* of *objects*
    - Changeable size
    - Different elements can hold different types
        - Every element is an "Object"
    - **Vectors** are "thread-safe"; **ArrayLists** are not (but are otherwise more efficient)

```
import java.util.Vector ;
. . .
Vector myPoints = new Vector ( );   // create an (empty) vector
. . .
Point p1 = new Point (5,3) ;       // create a Point named p1
myPoints.addElement (p1) ;         // adds a reference to Object
p1
```



- A common mistake:

```
. . .
myPoints.addElement (p1) ;    // add a point
p1.x ++ ;                     // modify the point
myPoints.addElement (p1) ;    // add a new, different point?  NO!
                              // (adds a second reference to
                              // SAME point (with modified value)
                              // (i.e. elementAt(0) is changed!!
```

Vectors and ArrayLists (cont.) :

- Advantages of Vectors/ArrayLists:
    - Can grow/shrink dynamically (automatically)
    - Can hold *different object types* in different elements
- Drawbacks of Vectors/ArrayLists:
    - Lose the familiar "indexing" syntax
        - **`myPoints[i]`** becomes **`myPoints.elementAt(i)`**
    - *Slight* space and time penalties over arrays
    - All elements are Objects (instances of Java class "Object")
        - Must type-cast to the appropriate type when retrieving

```
//create some Points and add them to myPoints Vector
. . .
Point p1 = new Point (6,4) ;
myPoints.addElement (p1) ;
. . .

//fetch the 'ith' Point from myPoints Vector
Point nextPoint = myPoints.elementAt (i) ;          // RUNTIME ERROR !!
Point nextPoint = (Point) myPoints.elementAt (i) ;     // correct way
```

<u>Vectors and ArrayLists (cont.) :</u>

- Vectors have *<u>many</u>* methods for manipulating elements (ArrayLists have similar – though not identical – list) :

```
add ( ) ;
addElement ( ) ;
clear ( ) ;
capacity ( ) ;
elementAt ( ) ;
equals ( ) ;
indexOf ( ) ;
insertElementAt ( ) ;
isEmpty ( ) ;
removeElementAt ( ) ;
size ( ) ;
. . .
```

- How do you know what methods exist? And their parameters? And how they work?

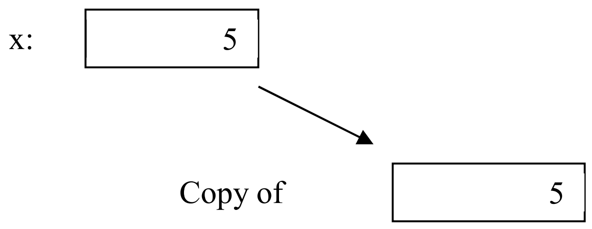
- Answer: **`https://docs.oracle.com/javase/8/docs/api/`**
  - Complete online Java Standard Edition 8 API documentation
  - Can be downloaded to your machine

PARAMETERS:

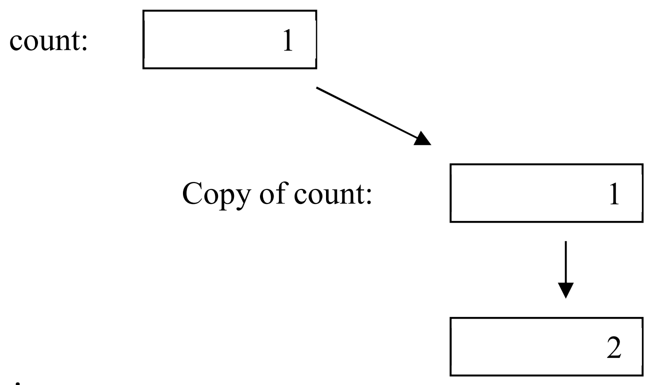
- ALL parameters are passed using "Call by Value" – NEVER "Call by Reference"
    - A <u>*copy*</u> of the actual value of the parameter is passed
- The original parameter CANNOT be modified by any method to which it is passed
- The *effect* of this *appears* to be different for **primitives** and **objects** (it's not)

Examples: Primitives

```
int x = 5 ;

int y = Math.sqrt (x) ;
```



```
. . .
int count = 1 ;
update (count) ;
System.out.println (count) ;
. . .

=============================

public void update (int count) {
      . . .
      count = count + 1 ;
      System.out.println (count) ;
}
```



PARAMETERS, cont:

- Parameters which are *objects* are also passed "by value" -- i.e. a *copy* is passed
- The original parameter still CANNOT be modified by any method to which it is passed
- But: objects are represented by *references:* a *<u>copy of the reference is passed</u>*
    - Result: the receiving method cannot alter the original *reference – but it <u>can</u> alter the object itself* (since it has a reference to it)

Examples: Objects:

```
Ball myBall = new Ball (Color.red);
. . .
System.out.println (myBall.color) ;     //presumes "color" is
                                        // public in Ball
display (myBall) ;
     System.out.println (myBall.color) ;
. . .

     =================================

public void display (Ball theBall) {
     System.out.println (theBall.color) ;
     theBall.color = Color.blue ;
}
```

Java vs. C++[†]

Java has:

- **No "preprocessor" (hence no `#include`, .h files, `#define`, etc…)**
- **No "global variables"**
- **Fixed (defined) sizes for primitives (e.g., `int` is *always* 32 bits)**
- **No Pointers. "References" are similar, but:**
  - **Cannot be converted to a primitive**
  - **Cannot be manipulated with arithmetic operators**
  - **Have no "&" (address-of) or dereference (" * " or " -> ") operators**
- **Automatic garbage collection**
  - **Objects which cannot be accessed (are "out of scope" and have no copied references) are automatically returned to the "heap"**
- **No "`goto`" statement**
- **No "`struct`" or "`union`" types**
- **No "function pointers" (although this can be simulated by passing objects which implement a given interface)**
- **No support for multiple inheritance of method implementation**
- **A weaker form of `templates` (called `generics`) based on a notion called `type erasure`**
- **No support for operator overloading**

[†] Excerpted from Java In A Nutshell, David Flanagan, O'Reilly

California State University, Sacramento
Department of Computer Science

Elements of Matrix Algebra

1. Definition and Representation

A *matrix* is a rectangular array of elements, arranged in rows and columns. We frequently number the rows and columns starting from zero, as shown:

$$\begin{array}{c|ccc} & 0 & 1 & 2 \\ \hline 0 & X & X & X \\ 1 & X & X & X \\ 2 & X & X & X \\ 3 & X & X & X \end{array}$$

We characterize a matrix by giving the number of rows, then columns: the example above is a *4x3* matrix. Note that by convention the number of *rows* is always given first.

In general the elements of a matrix can contain any object. However, when the elements are *numbers*, certain useful operations can be defined. The following sections describe some common matrix operations and assume the elements of the matrices in question are numbers.

2. Scalar Multiplication

One well-defined operation on a matrix is *scalar multiplication*, meaning multiplying a specific scalar value into a matrix. The result of scalar multiplication is to produce a new matrix with the same number of rows and columns as the original matrix, and where each element of the new matrix contains the product of the scalar value with the corresponding element value from the original matrix.

For example, the following shows the result of multiplying the scalar value 2 by the matrix M1, producing a new matrix M2:

$$2 * \overset{M1}{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}} = \overset{M2}{\begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix}}$$

3. Matrix Addition

Two matrices can be added together to produce a new (third) matrix. However, this operation is only defined if both the number of rows in the first matrix is the same as the number of rows in the second matrix and also the number of columns in the first matrix is the same as the number of columns in the second matrix.

For two matrices A and B which have the same number of rows and also the same number of columns, the sum A + B is a new matrix C where C has the same number of rows and columns as A and B, and where each element of C is the sum of the corresponding elements of A and B.

For example, the following shows the result of adding two matrices A and B:

$$\overset{A}{\begin{pmatrix} 1 & 5 \\ -2 & 4 \\ 3 & 1 \end{pmatrix}} + \overset{B}{\begin{pmatrix} 2 & 2 \\ -4 & 6 \\ -3 & 1 \end{pmatrix}} = \overset{C}{\begin{pmatrix} 3 & 7 \\ -6 & 10 \\ 0 & 2 \end{pmatrix}}$$

Note that because of the definition of the elements of C (being the sum of the corresponding elements of A and B), it is the case that matrix addition is *commutative*; that is, A + B = B + A.

4. Vector Multiplication

A matrix which has only one row is sometimes called a *vector*. (This is because of the similarity to the vector algebra representation for vectors – as a single-row arrangement of vector components.) For example, the following are two different vectors (single-row matrices):

$$\begin{bmatrix} 2 & 3 \end{bmatrix} \qquad \begin{bmatrix} 4 & 6 & -1 \end{bmatrix}$$

Given a vector (single-row matrix) and another matrix, the two can be multiplied together. However, this operation is only defined if the number of elements in the vector (the number of vector "columns") is equal to the number of *rows* in the matrix by which it is multiplied.

In that case, the result of the multiplication is a new *vector* (single-row matrix) with the same number of elements (columns) as the number of *columns* in the *matrix*, and where the value of each element of the new vector is the sum of the products of corresponding elements in the original vector and the corresponding column of the matrix. This operation is known as taking the *inner product* (also called the "***dot product***") of the vector with each matrix column. Note that it is the inner product of the vector with the first matrix column that forms the first element in the result vector, and so forth.

For example, the following shows the result of multiplying a 1x2 vector V1 with a 2x3 matrix M1, producing a new vector V2. Note that V1 has 2 columns and M1 has two rows, so they met the requirements for being able to perform the multiplication operation. Note also that the new vector (V2) has the same number of elements (3) and the number of *columns* in the *matrix*.

$$\begin{array}{ccccc} V1 & & M1 & & V2 \\ \begin{bmatrix} 2 & 3 \end{bmatrix} & * & \begin{bmatrix} 1 & 2 & -1 \\ 4 & 5 & 0 \end{bmatrix} & = & \begin{bmatrix} 14 & 19 & -2 \end{bmatrix} \end{array}$$

It is important to note that the result of multiplying a vector by a matrix is always a new *vector*, and that the new vector always has the same number of elements (columns) as the original *matrix.* The elements of V2 above were formed by computing each of the following inner products ("•" represents the inner product or "*dot product*" operation) :

[2 3] • [1 4] = (2*1) + (3*4) = 14 ;

[2 3] • [2 5] = (2*2) + (3*5) = 19 ; and

[2 3] • [-1 0] = (2*-1) + (3*0) = -2.

5. Row-major vs. Column-major Notation

The previous section describes a vector as a matrix with a single *row*; however, a vector can also be viewed as a matrix with a single *column*. In this case the vector is written vertically, with the elements forming a column – for example:

$$\begin{bmatrix} 2 \\ 3 \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} 4 \\ 6 \\ -1 \end{bmatrix}$$

The difference in the two representations is simply one of convenience; the column form (also called "column-major form") of a vector does not somehow represent a "different" vector than the equivalent row-major form.

However, along with the difference in notation comes a difference in representation of the multiplication operation. The multiplication of a (row) vector by a matrix always proceeds "from the left" – that is, the vector is always written on the left side of the multiplication sign (as shown in the preceding section), and the inner products are formed between the vector and consecutive *columns* of the matrix. When a vector is represented in "column-major" form, multiplication is always written with the vector on the *right* side, and the multiplication always proceeds from *right to left*.

For column-major representation, the elements of the result vector are formed by computing the inner products of the vector with each *row* of the matrix. This therefore means that for column-major representation (multiplication from the right), the number of elements (rows) of the vector must equal the number of *columns* in the matrix (as opposed to being equal to the number of rows in the matrix, which is the rule for row-major representation and multiplication from the left).

The form for column-major representation and multiplication from the right is shown below. Note that in this example the initial vector is written on the *right*, and the number of *columns* in the matrix matches the number of elements (rows) in the initial vector. Note also that the result (which is a vector, as before) is formed by computing the inner product of the initial vector with each *row* of the matrix (instead of with each column of the matrix as is done with row-major representation and multiplication

$$\overset{\text{V2}}{\begin{pmatrix} 14 \\ 19 \\ -2 \end{pmatrix}} = \overset{\text{M1}}{\begin{pmatrix} 1 & 4 \\ 2 & 5 \\ -1 & 0 \end{pmatrix}} * \overset{\text{V1}}{\begin{pmatrix} 2 \\ 3 \end{pmatrix}}$$

from the left).

The form which is used for vector multiplication (row/from the left or column/from the right) is mostly a matter of preference and notational convenience. Mathematics textbooks tend to use column-major form, while computer graphics texts tend to be split evenly between row-major and column-major form. Programming languages which support matrix operations tend to use column-major form. Regardless of which notation is used, it is important to be consistent, and it is also of critical importance to compute the inner products based on the representation given.

6. Transpose

The *transpose* of a matrix A is formed by "exchanging" the rows and columns of A such that for every element (i,j) in the original matrix, the same value is found at element (j,i) in the new matrix. That is, the value at row 2, column 1 is exchanged with the value at row 1, column 2, and so forth. This generates a new matrix called the *transpose* of A, denoted A^T. The following shows an example of a matrix A and its transpose A^T.

$$A = \begin{pmatrix} 2 & 10 & -3 \\ -4 & 8 & 4 \\ 6 & 2 & -7 \end{pmatrix} \qquad A^T = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \\ -3 & 4 & -7 \end{pmatrix}$$

Note that for a square matrix (such as A, above), transposing the matrix has the effect of "flipping" it along the diagonal running from upper left to lower right (called the "major diagonal'). However, a matrix need not be square to form the transpose, as shown in the next example:

$$B = \begin{pmatrix} 2 & 10 \\ -4 & 8 \\ 6 & 2 \end{pmatrix} \qquad B^T = \begin{pmatrix} 2 & -4 & 6 \\ 10 & 8 & 2 \end{pmatrix}$$

Note that transposing a non-square matrix has the effect of producing a new matrix whose number of *rows* is equal to the number of *columns* in the original matrix (and vice versa).

Another important point to note is that in the discussion of row-major vs. column-major representation (above), switching from multiplication on the left (with the vector on the left) to multiplication on the right (vector on the right) essentially involves forming the transpose of the matrix which is being multiplied (compare the examples in the two preceding sections to confirm this). That is, multiplying a row vector by a matrix "from the left" produces an equivalent result to multiplying the column form of the same vector "from the right" into the transpose of the matrix.

Two important identities relating matrix transposes are:

1. $(A + B)^T = A^T + B^T$; that is, the transpose of a sum of two matrices equals the sum of the transposes of the individual matrices.

2. $(A * B)^T = B^T * A^T$; that is, the transpose of a matrix product A*B is the product of the transpose of B times the transpose of A. Note that since matrix multiplication is *not* commutative (see below), the order of this result is important.

7. Matrix Multiplication

The generalized form of matrix multiplication is to multiply a matrix A by a second matrix B (with A on the left and B on the right) producing a third matrix C. However, this operation is only defined if the number of *columns* of the first matrix A is the same as the number of *rows* of the second matrix B. In such a case, matrices A and B are said to be *conforming* matrices.

The result of multiplying two conforming matrices A * B is a new matrix C which has the same number of *rows* as A and the same number of *columns* as B. That is, if A is an *m x p* matrix (i.e, has "m" rows and "p" columns), and B is a *p x n* matrix ("p" rows and "n" columns), then the product A*B produces a new matrix C which is *m x n*. Further, each element (i,j) of C is the scalar value produced by the inner product of the ith row of A with the jth column of B.

For example, multiplying the following *2 x 3* matrix A by the *3 x 4* matrix B produces the *2 x 4* matrix C as shown:

$$\underset{A}{\begin{pmatrix} 2 & 3 & -1 \\ 4 & 0 & 6 \end{pmatrix}} * \underset{B}{\begin{pmatrix} 1 & -1 & 0 & 4 \\ 2 & -2 & 1 & 3 \\ 5 & 7 & 1 & -3 \end{pmatrix}} = \underset{C}{\begin{pmatrix} 3 & -15 & 2 & 20 \\ 34 & 38 & 6 & -2 \end{pmatrix}}$$

Each element of C in the above example is formed by computing the inner product of a row of A with a column of B. For example, the element with value "3" in row 0, column 0 of C is formed by the inner product of row 0 of A with column 0 of B: (2*1) + (3*2) + (-1*5) = 3. Likewise, the element with value "6" in row 1 (the second row), column 2 (the third column) of C is formed by the inner product of row 1 of A with column 2 of B: (4*0) + (0*1) + (6*1) = 6. Each of the other elements of C is likewise formed by computing the inner product of a row of A with a column of B.

Note that because the matrix multiplication operation is only defined when the number of columns of the first (left) matrix equals the number of rows of the second (right) matrix, in general it is *not* true that just because A*B is a defined operation then it follows that B*A is also well defined. In the example above, for instance, A*B is defined (and produces the matrix C as shown), but the operation B*A *is not defined* – because B does not have the same number of columns as the number of rows in A.

The only time that both A*B and B*A are well-defined matrix multiplication operations is when both A and B are *square* matrices of the *same size*.

Note that just because A and B are square matrices of the same size (and hence both A*B and B*A are well-defined), it is not in general true that A*B = B*A. That is, *the matrix multiplication operation is not commutative*. This property (lack of commutativity of matrix multiplication) is important to keep in mind when manipulating matrices.

8. Identity Matrix

We define a special matrix form called the *Identity matrix*. The Identity matrix has the properties that (1) it is square; (2) it has 1's in every element along its major diagonal; and (3) it has zeroes in every element not on its major diagonal. For example, the following shows an Identity matrix I of size 3:

$$I = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Identity matrices have certain important and useful properties. For example, if I is an identity matrix of size *n*, and A is also a square matrix of size *n*, then it is the case that

$$A * I = I * A = A.$$

That is, multiplying matrix A by the identity matrix, whether on the left or on the right, leaves A unchanged (this is in fact the mathematical definition of an "identity element"; for example, the value "1" is the identity element with respect to the ordinary algebra operation "multiplication" – multiplying a given number by 1 does not change its value).

9. Matrix Inverses

A given matrix "M" is said to be *invertible* (or "*has an inverse*"), if there exists another matrix, denoted as M^{-1}, such that

$$M * M^{-1} = M^{-1} * M = I,$$

where I is the Identity matrix of the same size as M. The following general properties relate to matrix inverses:

(1) only *square* matrices have inverses (this is a consequence of the above definition, which requires the same result for multiplication from the left and right);

(2) not all square matrices have inverses (that is, there are some matrices for which, even though they are square, it is not possible to find another matrix for which the above equations hold); and

(3) if a matrix has an inverse, it will be *unique* (that is, every matrix has at most one inverse).

An important observation about matrix inverses is that they represent, in a sense, an "opposite" operation. That is, if M represents some operation (say, a transformation of some kind), then M^{-1} represents the "opposite transformation". This observation is useful when attempting to build a series of transformations which are represented by matrices; it provides a method of specifying how to "undo" a given operation. However, care must be taken to insure that the operation in question (being represented by a matrix) is "undoable" (i.e. that the matrix is invertible). See the section on "Inverse Of Products" for further information on this topic.

10.Associativity and Commutativity

As noted above, the matrix multiplication operation is *not* commutative. That is, it is *not* true (in the general case) that A*B produces the same result (matrix) as B*A.

However, an important property of matrix multiplication is that it is *associative.* That is, given three matrices A, B, and C, multiplied from left to right, the same result will be obtained whether the left or right multiplication is performed first. That is,

$$(A * B) * C = A * (B * C)$$

This associative property of matrix multiplication allows matrix operations to be combined in various different ways for efficiency.

11.Inverse of Products

A useful theorem for manipulating matrices is that *the inverse of a matrix product is equal to the product of the inverses of the individual matrices, multiplied in the opposite order.* For example, suppose we have the matrix product (A * B * C), which as noted above can be computed as ((A * B) * C) or (A * (B*C)) due to associativity of matrix multiplication. The inverse of this product is denoted $(A * B * C)^{-1}$. Then the above theorem says that

$$(A * B * C)^{-1} = C^{-1} * B^{-1} * A^{-1}.$$

That is, the inverse of (A*B*C) can be obtained by first obtaining each individual inverse matrix (A^{-1}, B^{-1}, and C^{-1}), and then multiplying those matrices together in the opposite order.

Note that the product on the right-hand side is written in the opposite order from the one on the left, and that *this order is significant since matrix multiplication is **not** commutative* (even though it is associative).

The inverse-of-products theorem is useful in situations where you have a series of transformations represented in matrix form and you want to "undo" the transformations – that is, you want to find a transformation which goes in the "opposite direction". If the original series of transformations is A, then B, then C, then the "opposite" transformation would be that which "undoes (A, then B, then C)" – that is the *inverse* of (A * B * C). By the inverse-of-products theorem, this "opposite" transformation is exactly the transformation ($C^{-1} * B^{-1} * A^{-1}$). (Note: in order for this to work, it must be true that each individual matrix (A, B, and C) is itself invertible.

12.Order of Application of Matrix Transforms in Code

Matrices can be used to represent "transformations" to be applied to objects. For example, a single matrix can represent a "translation" operation to be applied to a "point" object. Applying matrix transformations correctly in program code requires a thorough understanding of the rules of matrix algebra regarding associativity, commutativity, and inverse products (discussed above), as well as an understanding of how code statements translate into matrix operations.

Suppose for example that you wish to apply a translation, represented by a matrix, to a given point P1. This is done by multiplying the point (represented as a vector as described above) by the matrix, which results in a new (translated) point P2. As noted above, matrix multiplication can be done either using “row-major” form or “column-major” form. However, most programming language matrix libraries (including those in Java) use column-major form – that is, matrix multiplication is done “from the right”. Assuming we are using this convention (for example, assuming we are writing in Java), then when you multiply a point by a transformation matrix, you do so "from the right", meaning that the point is written on the right hand side with the matrix to its left, and the multiplication proceeds that way ("from right to left").

The algebraic representation of the above example would be the following, where “P1” is the original point and “M” is the matrix containing the desired translation:

$$\overset{P2}{\begin{bmatrix} X' \\ Y' \end{bmatrix}} = \overset{M}{\begin{bmatrix} \quad & \quad \\ \quad & \quad \end{bmatrix}} * \overset{P1}{\begin{bmatrix} X \\ Y \end{bmatrix}}$$

Suppose for example that the original point P1 was (1,1) and the matrix contained a translation of (1,1). Then the Java code to accomplish this would be:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.translate(1, 1);

Point p2;
at.transform(p1,p2);
```

“**`AffineTransform`**” is the Java class representing transformation matrices; in fact, we sometimes for convenience just refer to objects of type **`AffineTransform`** as “matrices”. **`translate()`** is a method which causes the specified matrix (**`AffineTransform`** object) to contain the specified translation, and **`transform()`** is a method which multiplies its first argument by the matrix and returns the result in the second argument, with the multiplication being applied “from the right”. In the above example the original point P1 has a value of (1,1) and the matrix has a translation of (1,1), so the value of P2 after executing the above code will be (2,2) since applying a translation of (1,1) to a point with value (1,1) results in a point with value (2,2).

A transformation matrix contains (in the general case) multiple transforms -- e.g. a scale, a rotate, a translate, another rotate, etc. These transformations exist (conceptually) in the matrix *in some order* -- that is, there is one which is "rightmost", one immediately to its left, one immediately to THAT one's left, etc. The order in which the transformations exist in the matrix is determined by the code which inserts them into the matrix. Inserting a transformation is done by multiplying the new transformation “on the right” of the existing transformation, so the new transformation exists “on the right” in the new compound transformation matrix.

For example, the following Java code creates a transformation matrix (**`AffineTransform`** object) containing TWO transformations: a rotation by 90° and a translation by (1,1):

```
AffineTransform at = new AffineTransform();
at.translate(1, 1);
at.rotate(Math.toRadians(90));
```

Initially (that is, when it is first created) the matrix contains the Identity transformation. Since the first method invoked on the matrix is `translate()`, the specified translation becomes the "leftmost" (and in this case the only) transformation in the matrix. Since the `rotate()` is called *after* the `translate()`, the rotation is concatenated "on the right" in the matrix; that is, the rotation is the rightmost transformation in the matrix and the translation is to the left of the rotation.

When you multiply a point by a matrix, you are applying the transformations contained in the matrix to the point "in order, from right to left". That is, the rightmost transformation gets applied first, then the one to its left, then the next leftward one, etc. For example, consider the following Java code:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.translate(1, 1);
at.rotate(Math.toRadians(90));

Point p2;
at.transform(p1,p2);
```

The value in P2 after this code is executed will be (0,2). This is because the `transform()` method multiplies the point P1 by the matrix *from the right*, causing the point to first be transformed by the rightmost transformation in the matrix (the rotation) – which rotates the original point (1,1) to the point (-1,1) – then transformed by the translation, which translates the point (-1,1) by (1,1) resulting in the value (0,2) in P2. (If this is not clear you should draw the points on graph paper, construct the matrix containing the two transformations by hand, and convince yourself that the above statements are correct.)

Note that it is the case (as shown in the example above) that *the order of **application** of transformations contained in a matrix is the **opposite of the order in which the code statements defining the transformations are executed***. In the above example the `translate()` method was called *before* the `rotate()` method, meaning that when the `transform()` method was invoked to multiply the point by the matrix the ***rotation*** was applied *before* the translation.

Note that if the above example had instead been written as:

```
Point p1 = new Point(1,1);
AffineTransform at = new AffineTransform();
at.rotate(Math.toRadians(90));
at.translate(1, 1);

Point p2;
at.transform(p1,p2);
```

(that is, if the rotation had been concatenated into the matrix *first*), then the result would have been different – specifically, the final value in P2 would have been (-2,2), since the `transform()` method would have the effect of applying the translation first (translating the original point (1,1) to (2,2)) and then applying the rotation second (rotating the point (2,2) to (-2,2)). Note also that this difference is a manifestation of the algebraic rule that matrix multiplication is not commutative, as discussed above.

Therefore, if you have a certain order in which you want a set of tranformations *applied*, you must get them into the transformation matrix such that they are in order, from *right* to *left*, in the order that you want them applied. If you have, say, a scale and a translate to be applied, and you want the translate to be applied first, it must be on the rightmost side of the matrix, with the scale to its left. This in turn means that you must put the SCALE in the matrix *first*, and THEN put the translate in the matrix so that the translate is on the right (and the scale to its left). This means you must call the AffineTransform `scale()` method in your code BEFORE you call the AffineTransform `translate()` method. Writing code to apply matrix transformations therefore involves first figuring out the order in which you want the transforms applied, and then writing code statements (in the proper order) that cause those transforms to end up in the matrix in the right-to-left order you need.

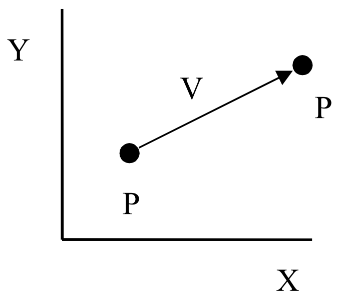
California State University, Sacramento
Department of Computer Science

Elements of Vector Algebra

1. Definition and Representation

A *vector* is an object with *magnitude* and *direction.* A vector is commonly represented as an arrow, with the length of the arrow representing the magnitude, and the direction of the arrow representing the vector direction. The starting point of the arrow is called the *tail* of the vector; the ending point (arrowhead) is called the *head* of the vector.

A vector runs from one point to another, and can be represented as the *difference* between the two points. The "difference between two points" is obtained by taking the difference between corresponding elements of the two points. Thus, in 2D:



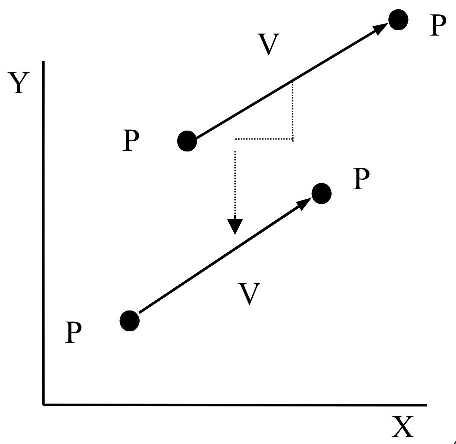
$V = (P2 - P1)$

$= [\ (X_2 - X_1)\ \ (Y_2 - Y_1)\ ]$

Note that a vector is represented as an object in square brackets; the elements inside the square brackets are called the *components* of the vector. Each component of a vector is a numerical quantity.

2. Translation

Vectors can be *translated* without altering their value (since they consist of *magnitude* and *direction* but not *position*):



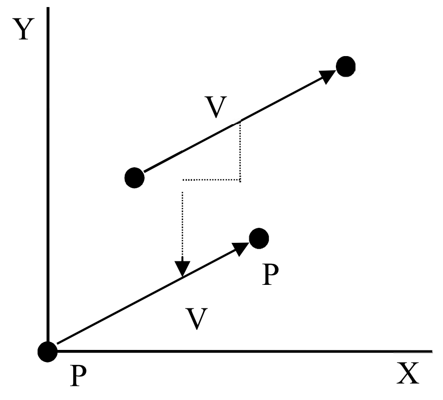
$V1 = (P2 - P1)$

$V2 = (P4 - P3)$

$P4 - P3 = P2 - P1;$

hence $V2 = V1$

Since a vector can be translated anywhere without changing its value, it follows that a vector can be translated so that its tail is at the origin:

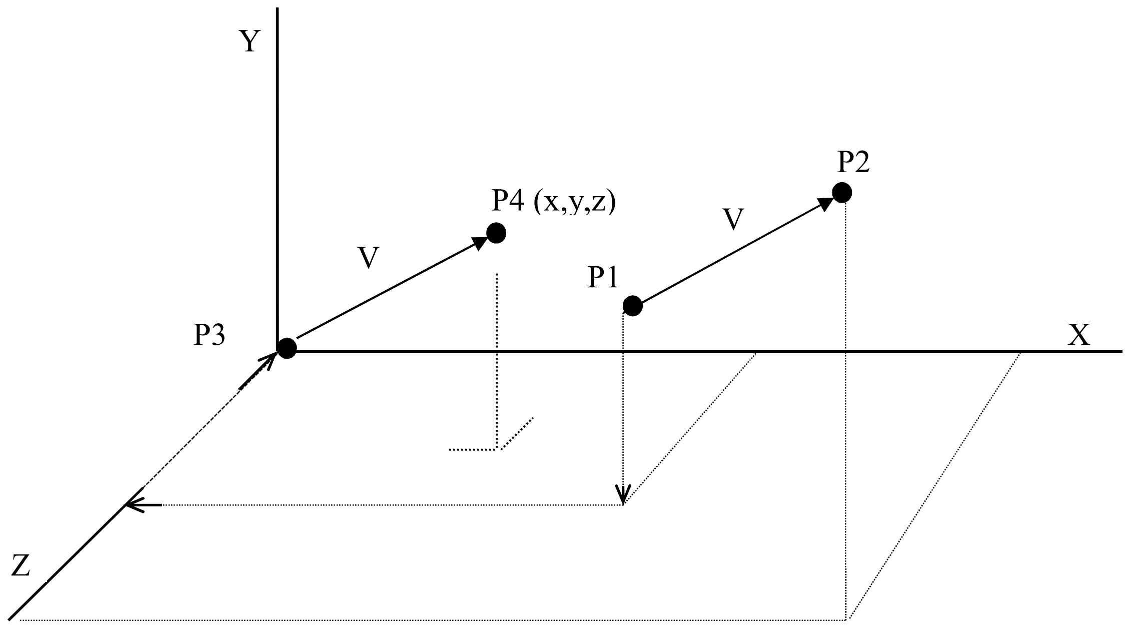


$$
\begin{aligned}
V &= (P2 - P1) \\
&= [\ (X_2 - X_1)\ \ (Y_2 - Y_1)\ ] \\
&= [\ (X_2 - 0)\ \ (Y_2 - 0)\ ] \\
&= [\ (X2)\ \ (Y2)\ ] \\
&= [\ X2\ \ Y2]
\end{aligned}
$$

Thus, a vector can also be represented as a *single point*: the point at the head of the vector when the tail is at the origin. Both representations of vectors (as the difference of two points or as a single point) are useful.

3. Two-Dimensional vs. Three-Dimensional Vectors

Vectors in 3D work the same way as in 2D. They can be specified as the difference between two (3D) points; they can be translated without changing their value; and they can be specified as a single (3D) point since the tail can be translated to the origin. (In fact, a 2D vector can be considered to be a special case of a 3D vector, with the Z component equal to zero.)



In the preceding figure,

$$V = (P2 - P1)$$

$$= (P4 - P3)$$

$$= [\ (X_2 - X_1)\ \ (Y_2 - Y_1)\ \ (Z_2 - Z_1)\ ]$$

$$= [\ (X_4 - X_3)\ \ (Y_4 - Y_3)\ \ (Z_4 - Z_3)\ ]$$

$$= [\ (X_4 - 0)\ \ (Y_4 - 0)\ \ (Z_4 - 0)\ ]$$

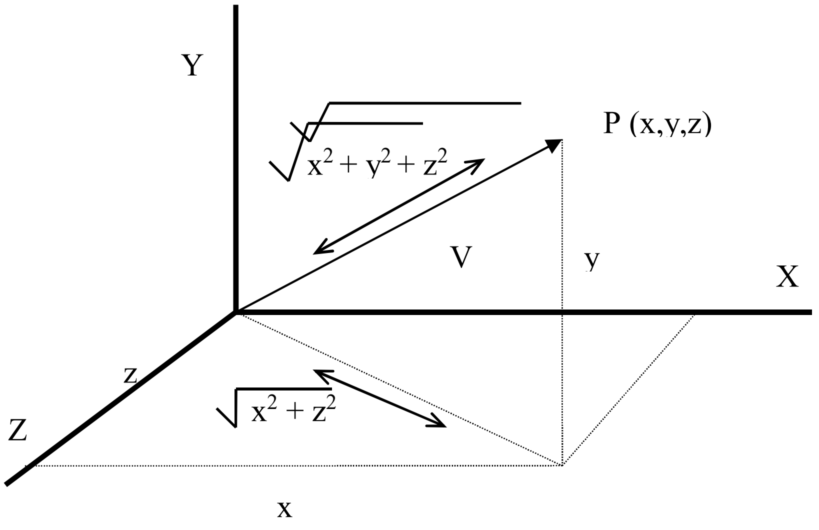
$$= [\ X_4\ \ Y_4\ \ Z_4\ ]$$

$$= [\ x\ \ y\ \ z\ ]$$

Note therefore that a *single point* (x, y, z) in 3 dimensions represents the 3D vector **[x y z]** , the vector from the origin to (x, y, z).

4. Magnitude

The *length* or *magnitude* of a vector V is denoted by the symbol |V|, and can be calculated using the Pythagorean Theorem:

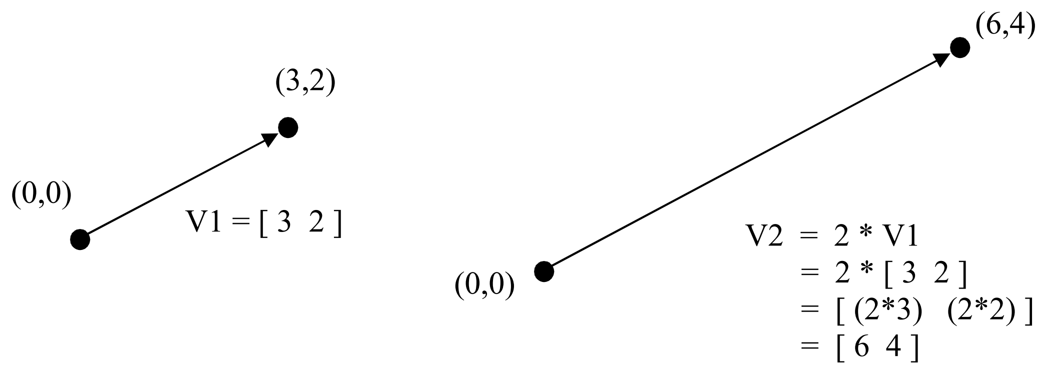


The magnitude (length) of V is |V| = $(x^2 + y^2 + z^2)$.

Note again that we can consider 2D a special case of 3D where Z=0; in that case the magnitude of a 2D vector is just |V| = sqrt $(x^2 + y^2)$.

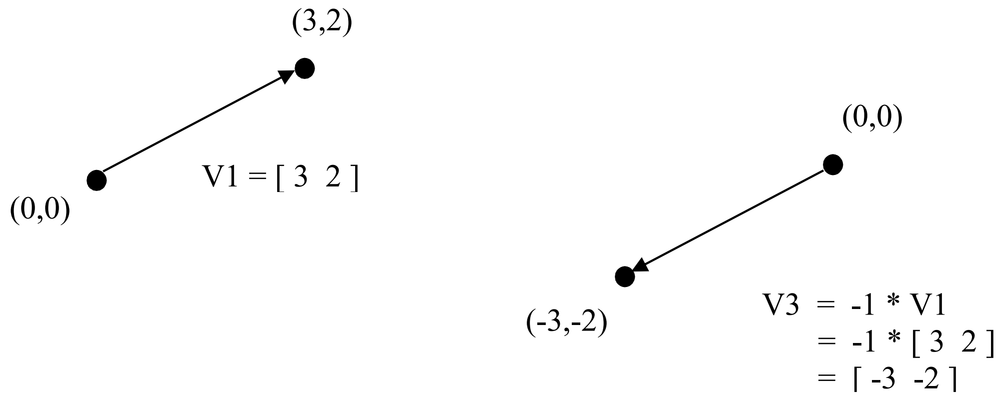
5. Scalar Multiplication

Multiplying a vector by a scalar value is a well-defined operation, and produces a new vector whose components are the result of multiplying each of the original vector components by the scalar value. For example, multiplying the 2D vector V1 = [3 2] by the scalar value 2 produces the new vector V2 = [6 4]. This is shown graphically in 2D as:



Note that the effect of multiplying a vector by a positive scalar value is to produce a new vector whose direction is the same as the original vector and whose magnitude varies according to the scalar value: multiplying by a value greater than one increases the magnitude, while multiplying by a fractional value decreases the magnitude.

Note also that multiplying a vector by a *negative* scalar value has the effect of reversing the direction of the vector. For example, using the vector V1 = [3 2] (as shown above), a new vector V3 = -1 * V1 is [-3 -2], as shown below (keep in mind that vectors can be translated without changing their value) :



6. Unit Vectors

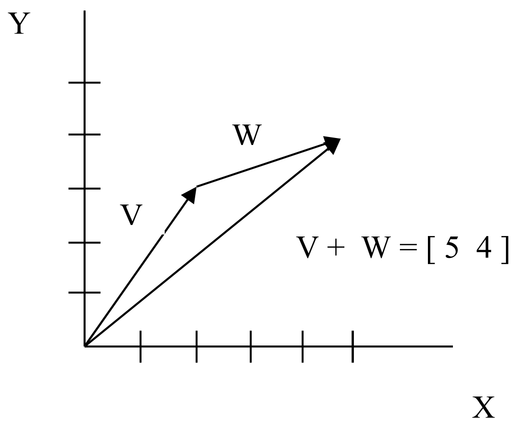
Given a vector V, there exists a corresponding vector U (also sometimes denoted as V with a "^" over it) called a *unit vector* with the property than U has the same *direction* as V but has *length = 1.*

The unit vector U for a given vector V is calculated by dividing each of the components of V by the magnitude (length) of V: $U = V / |V| = [\ (V_x / |V|)\ \ (V_y / |V|\ ]$. For example, if $V = [\ 3\ \ 4\ ]$, then $|V| = \text{sqrt}(\ 3^2 + 4^2) = 5$, and $U = [\ (3/5)\ \ (4/5)\ ] = [\ 0.6\ \ 0.8\ ]$. Note that this is a vector in the same direction as V, and whose length is $|U| = \text{sqrt}\ (0.6^2 + 0.8^2) = \text{sqrt}\ (0.36 + 0.64) = \text{sqrt}\ (1.0) = 1.0$.

7. Vector Addition

Given two vectors V and W, the vector sum V+W is a well-defined operation and produces a new vector whose components are the sums of the corresponding components of V and W. For example, if V = [2 3] and W = [3 1], then the vector V+W = [(2+3) (3+1)] = [5 4].

Vector addition can be represented graphically by placing the tail of one vector at the head of the other; the sum will be the vector from the tail of the first vector to the head of the second. For example, using the vectors V and W given above:



8. Dot Product

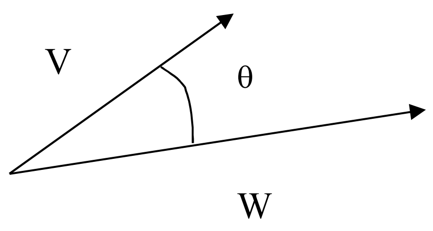
Another well-defined operation on vectors is called the *dot product* (also called the *inner product).* Given two vectors V and W, the dot product is written notationally as $V \bullet W$ and is defined as a scalar value $D = \Sigma\ (V_i * W_i)$ over all components "i" of V and W. Note that the dot product operation produces a *scalar* value D – i.e., a single numeric value.

For example, if V = [2 3] and W [7 -1], then the dot product

$V \bullet W = (\ (2 * 7) + (3 * -1)\ ) = (14 - 3) = 11.$ As noted, this result is a scalar value.

An interesting and useful property of the dot product of two vectors is that it produces the same value as the product of the magnitudes of the two vectors multiplied by the cosine of the smaller of the angles between the two vectors. That is, given two vectors V and W,

$V \bullet W = |V| * |W| * \cos(\theta)$, where θ is the angle between V and W:



Since both the magnitudes of the vectors and the scalar value of the dot product of the vectors can be easily calculated (see above), this means it is straightforward to find the angle between two vectors:

$$\cos(\theta) = (V \bullet W) \;/\; (|V| * |W|)$$

Note that if V and W are *normalized* (that is, they are Unit Vectors of length 1), then this formula reduces to

$$\cos(\theta) = (V \bullet W)$$