ENCE 688R Civil Information Systems

The Java Language

Mark Austin

E-mail: austin@isr.umd.edu

Department of Civil and Environmental Engineering, University of Maryland, College Park

Lecture 3: Topics

Part 1: Basic Stuff

- Primitive Data Types, Variables, Constants, Scope of a Variable.
- Arithmetic Operations and Expressions
- Control Statements
- Package and Import Statements

Part 2: Methods

- Syntax for defining a method.
- Polymorphism of methods.

Part 3: Working with Arrays

- One- and two-dimensional arrays.
- Ragged arrays.

Basic Stuff

Primitive Data Types

Primitive Data Types - Boolean, Char, 4 Integer Formats

Type	Contains	Default Value	Size	Range and Precision
boolean	True or false	false	1 bit	
char	Unicode character	\u0000	16 bits	\u0000 / \uFFFF
byte	Signed integer	0	8 bits	-128/127
short	Signed integer	0	16 bits	-32768/32767
int	Signed integer	0	32 bits	-2147483648/2147483647
long	Signed integer	0	64 bits	-9223372036854775808 / 9223372036854775807

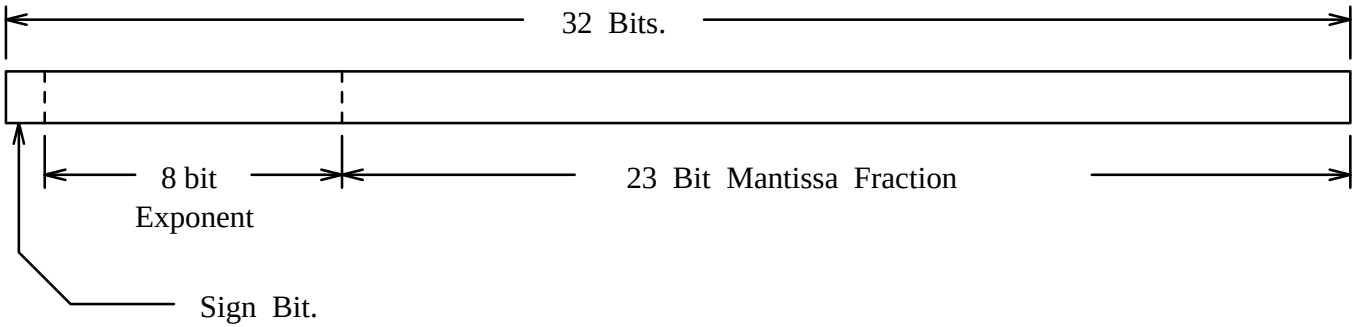
Primitive Data Types

Primitive Data Types – Two Formats for Float-Point Numbers

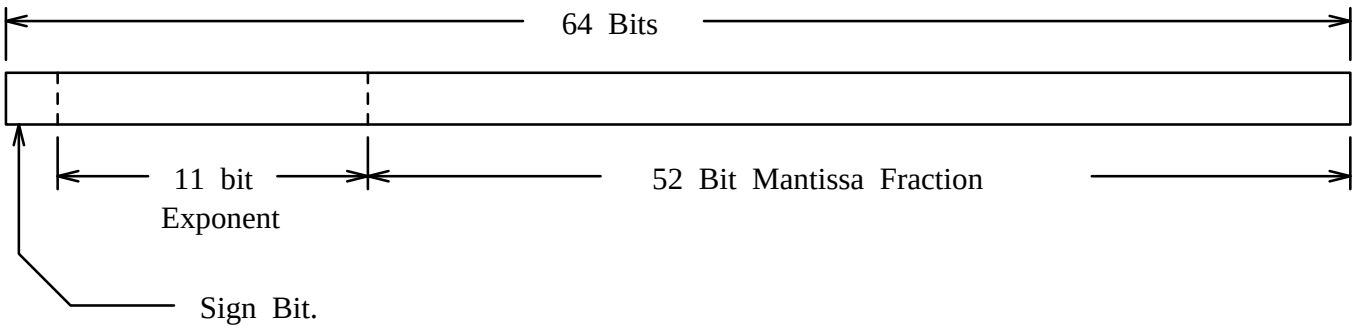
Type	Contains	Default Value	Size	Range and Precision
float	IEEE 754 floating point	0.0	32 bits	+- 13.40282347E+38 / +- 11.40239846E-45
	Floating point numbers are represented to approximately 6 to 7 decimal places of accuracy.			
double	IEEE 754 floating point	0.0	64 bits	+- 11.79769313486231570E+308 / +- 14.94065645841246544E-324
	Double precision numbers are represented to approximately 15 to 16 decimal places of accuracy.			

IEEE 754 Floating Point Standard

Layout of Memory



IEEE FLOATING POINT ARITHMETIC STANDARD FOR 32 BIT WORDS.



IEEE FLOATING POINT ARITHMETIC STANDARD FOR DOUBLE PRECISION FLOATS.

IEEE 754 Floating Point Standard

Support for Run-Time Errors

This standard includes:

- Positive and negative sign-magnitude numbers,
- Positive and negative zeros,
- Positive and negative infinites, and
- Special Not-a-Number (usually abbreviated NaN).

NaN value is used to represent the result of certain operations such as dividing zero by zero.

Java Variables

Definition

A **variable** is simply ...

... a block of memory whose value can be accessed with a name or identifier.

A variable contains either the contents of a primitive data type or a reference to an object. The object may be...

... an instance of a class, an interface, or an array.

Four Attributes of a Variable

- A type (e.g., int, double, float),
- A storage address (or location) in computer memory,
- A name, and
- A value.

All four parts must be known before a variable may be used in a program.

Java Variables

Variable Declarations

Variables must be declared before they can be used, e.g.,

```
int     iA =  10;
float   fA = 0.0;
double 8dA = 0.0;  <--- illegal! Cannot begin a variable name with
                        a digit.
```

What happens at compile and run time?

When a compiler encounters a variable declaration, ..

1. It will enter the variable name and type into a symbol table (so it knows how to use the variable throughout the program).
2. It generate the necessary code for the storage of the variable at run-time.

Three Types of Java Variable

Local Variables

- These are variables whose scope is limited to a block of code.
- Local variables are defined within the current block of code and have meaning for the time that the code block is active.

An Example

```
Source code                                              Output
=========================================================  ================
for ( int i = 0; i <= 2; i = i + 1)                        Loop 1: i = 0
   System.out.println( "Loop 1: i = " + i );               Loop 1: i = 1
                                                           Loop 1: i = 2
for ( int i = 0; i <= 2; i = i + 1)                        Loop 2: i = 0
   System.out.println( "Loop 2: i = " + i );               Loop 2: i = 1
                                                           Loop 2: i = 2
```

Three Types of Java Variable

Instance Variables

- These variables hold data for an instance of a class.
- Instance variables have meaning from the time they are created until there are no more references to that instance.

An Example

```
Definition of a class                    Using the class
=====================================    ==========================

public class Complex {                   Complex cA = new Complex();
   double dReal, dImaginary;             cA.dReal = 1.0;
   ....
}                                        Complex cB = new Complex();
                                         cB.dReal = 1.0;
=====================================    ==========================
```

Variables cA.dReal and cB.dReal occupy different blocks of memory.

Three Types of Java Variable

Class Variables

- These variables hold data that can be shared among all instances of a class.
- Class variables have meaning from the time that the class is loaded until there are no more references to the class.

An Example

```
Definition of a class                    Accessing the variable
=====================================    ==========================
public class Matrix {                    int i = Matrix.iNoColumns;
   public static int iNoColumns = 6.

   .....
}
=====================================    ==========================
```

The variable is static – no need to create an object first.

Java Variable Modifiers

Variable Modifiers

Modifier	Interpretation in Java
public	The variable can be accessed by any class.
private	The variable can be accessed only by methods within the same class.
protected	The variable can also be accessed by subclasses of the class.
static	The variable is a class variable.

Constants

Setting up constants

In Java constants are defined with ..

... variable modifier final indicating the value of the variable will not change.

An Example

```
Definition of a class                    Accessing the variable
=====================================    ==========================
public class Math {                            double dPi = Math.PI;
   public static final double PI = 3.14..;
   .....
}
=====================================    ==========================
```

The variable PI is both static and final. This makes PI a class variable whose assigned value cannot be changed.

Arithmetic Operations

Standard Arithmetic Operations on Integers and Floats

```
+       -       *       /
```

Modulo Operator

The modulo operator

```
%
```

applies only to integers, and returns the remainder after integer division. More precisely, if a and b are integers then

```
a % b = k*b + r
```

A Note on Integer Division

Integer division truncates what we think of as the fractional components of all intermediate and final arithmetic expressions, e.g.,

```
iValue = 5 + 18/4;      ===> 5 + 4  <=== Step 1 of evaluation
                        ===> 9      <=== Step 2 of evaluation
```

Probably not what we want!

Evaluation of Arithmetic Expressions

Hierarchy of Operators

Operator	Precedence	Order of Evaluation
() [] -> .	1	left to right
! ++ -- + -	2	right to left
* / %	3	left to right
+ -	4	left to right
<< >>	5	left to right
< ≤ > ≥	6	left to right
== !=	7	left to right
&	8	left to right
∧	9	left to right
|	10	left to right

Evaluation of Arithmetic Expressions

Hierarchy of Operators

Operator	Precedence	Order of Evaluation
&&	11	left to right
||	12	left to right
? :	13	right to left
= += *= /= &=	14	right to left
^= |= <<= >>=		
,	15	left to right

Dealing with Run-Time Errors

Dealing with Run-Time Errors

Source code

```
=================================================================
  double dA = 0.0;
  System.out.printf("Divide by zero:  ( 1/0.0) = %8.3f\n",  1.0/dA );
  System.out.printf("Divide by zero:  (-1/0.0) = %8.3f\n", -1.0/dA );
  System.out.printf("  Not a number: (0.0/0.0) = %8.3f\n",   dA/dA );
```

Output

```
=================================================================
Divide by zero:  ( 1/0.0) = Infinity
Divide by zero:  (-1/0.0) = -Infinity
  Not a number: (0.0/0.0) =      NaN
=================================================================
```

Dealing with Run-Time Errors

Print Variables containing Error Conditions

```
Source code
=================================================================
 double dB = 1.0/dA;
 System.out.printf("dB = 1.0/dA = %8.3f\n",  dB );
 double dC = dA/dA;
 System.out.printf("dC =  dA/dA = %8.3f\n",  dC );

Output
=================================================================
dB = 1.0/dA = Infinity
dC =  dA/dA =      NaN
=================================================================
```

Dealing with Run-Time Errors

Evaluate a Function over a Range of Values

```
System.out.println("Evaluate y(x) for range of x values");
System.out.println("==================================");

for ( double dX = 1.0; dX <= 5.0; dX = dX + 0.5 ) {
      double dY = 1.0 + 1.0/(dX - 2.0) - 1.0/(dX - 3.0) + (dX-4.0)/(dX-4.0);
      System.out.printf(" dX = %4.1f   y(dX) = %8.3f\n", dX, dY );
}
```

```
Evaluate y(x) for range of x values
==================================
 dX =  1.0   y(dX) =    1.500
 dX =  1.5   y(dX) =    0.667
 dX =  2.0   y(dX) = Infinity
 dX =  2.5   y(dX) =    6.000
 dX =  3.0   y(dX) = -Infinity
 dX =  3.5   y(dX) =    0.667
 dX =  4.0   y(dX) =      NaN
 dX =  4.5   y(dX) =    1.733
 dX =  5.0   y(dX) =    1.833
```

Dealing with Run-Time Errors

Test for Error Conditions

Source code

```
=====================================================================
if( dB == Double.POSITIVE_INFINITY )
    System.out.println("***  dB is equal to +Infinity" );

if( dB == Double.NEGATIVE_INFINITY )
    System.out.println("***  dB is equal to -Infinity" );

if( dB == Double.NaN )
    System.out.println("***  dB is Not a Number" );
```

Output

```
=====================================================================
***  dB is equal to +Infinity
***  dB is not equal to -Infinity
***  dB is Not a Number
=====================================================================
```

Control Statements

Control Statements

Control structures allow a computer program to ...

> **... take a course of action that depends on the data, logic, and calculations currently being considered.**

Machinery:

- Relational and logical operands;
- Selection constructs (e.g., if statements, switch statements).
- Looping contructs (e.g., for loops, while loops).

Common Error. Writing ...

```
if ( fA = 0.0 ) .....
```

instead of

```
if ( fA == 0.0 ) .....
```

Package Statements

Purpose of Packages

- Every class is part of a **package**, and every package is identified by its name.
- Packages provide ...

  ... a high-level layer of access protection and name-space management for collections of Java classes, interfaces, exceptions, and errors.

- Packages reduce the likelihood of name clashes because class and interface names are evaluated with respect to the package to which they belong.
- A package may include other packages (i.e., subpackages).

Simple Example

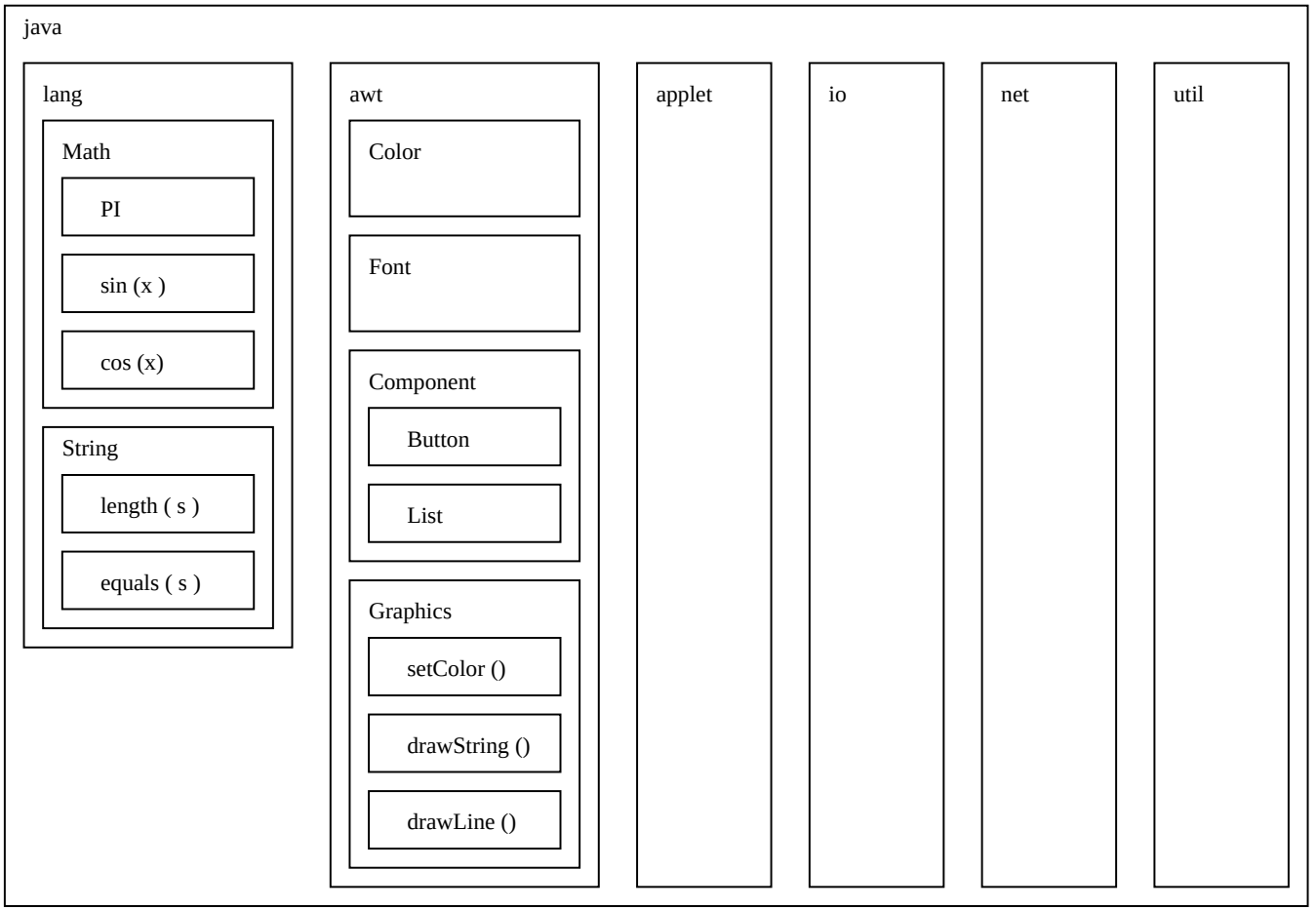
The statement

```
package fruit;
```

defines a package called **fruit**. There needs to be ...

... a one-to-one correspondence between the package name and a hierarchy of folders containing the Java source code.

Core Packages in Java



Import Statements

Import Statements

An import statement ...

... makes Java classes available to a program under an abbreviated name.

Import statements come in two forms:

```
import package.class;
import package.*;
```

The first form allows a class to be referred to by its class name alone. The asterisk (*) in the second form references all the classes in the named package.

Importing classes from java.lang.System

The java.lang.System package is so fundamental to Java programming that it is automatically imported into every Java program.

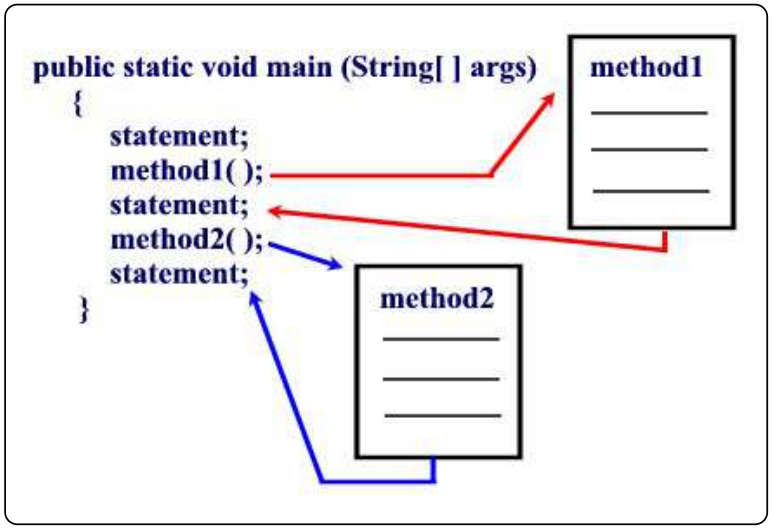
Methods

Definition of Methods

Formal Definition of Method

A method is a set of code which is ...

... referred to by name and can be called (invoked) at any point in a program simply by utilizing the method's name.



It is convenient to think of a method as a subprogram that acts on data and often returns a value. Each method has its own name.

Elements of a Java Method

Name

- All Java methods will have a name.

Argument List

- Most methods in Java will pass information from the calling method via an argument list.
- Occasionally we will encounter methods that have empty (void) argument lists.

Return Value

- Most of the Java methods we will encounter will return information to the calling method via the return value.
- Occasionally we see functions that do not return a data type (void return type).

Syntax for Defining a Method

Syntax for Defining a Method

The syntax for making a method definition in Java is

```
modifier return-type name-of-method ( parameter-list ) {

    ... executable statements ....

}                                  <=== end of the method body.
```

Key points:

- The **modifier** establishes ...

  ... the method type and its scope (i.e., what other methods can call it).

- The **return-type** specifies the type of information the method will return.
- Methods that do not return anything should use the return type `void`.

Class and Method Modifiers

Modifier	Interpretation in Java
abstract	The method is provided without a body; the body will be provided by a subclass.
final	The method may not be overridden.
native	The method is implemented in C or in some other platform-dependent way. No body is provided.
private	The method is only accessible within the class that defines it. You should use this keyword for methods that are only of concern to the internal details of the class.
public	The method is accessible anywhere the class is accessible.
static	Only one instance of a static member will be created, no matter how many instances of the class are created. These member functions may be accessed through the same class name.

Passing Arguments to Methods

Pass-By-Value Mechanism (for basic data types)

Java passes ...

> **... all primitive data type variables and reference data type variables to a method by value.**

A copy of the variable's value is used by the method being called.

Example

See the TryChange example on the class web site.

Polymorphism of Methods

Definition of Polymorphism

Polymorphism is ...

> **... the capability of an action to do different things based on the details of the object that is being acted upon.**

This is the third basic principle of object oriented programming.

Polymorphism of Methods

See the DemoPolymorphism program on the class web site. o

```
public static void doSomething() { ....

public static void doSomething( float fX ) { ....

public static void doSomething( double dX ) { ....
```

Three versions of a method with the same name!

Class Methods

Definition of Class Methods

A class method is a method that ...

> **... does not require an object to be invoked.**

Class methods are ...

> **... called in the same manner as instance methods except that the name of the class is substituted for the instance name.**

A Few Examples

Two of the most commonly used class methods are `System` and `Math`, e.g.,

```
System.out.println("Here is a line of text ...");

double dAngle = Math.sin( Math.PI );
```

Working with Arrays

Working with Arrays

Definition of an Array

In Java, an array is simply ...

... a sequence of numbered items of the same type.

Permissible types include:

- Primitive data types, and
- Instances of a class.

In either case, ...

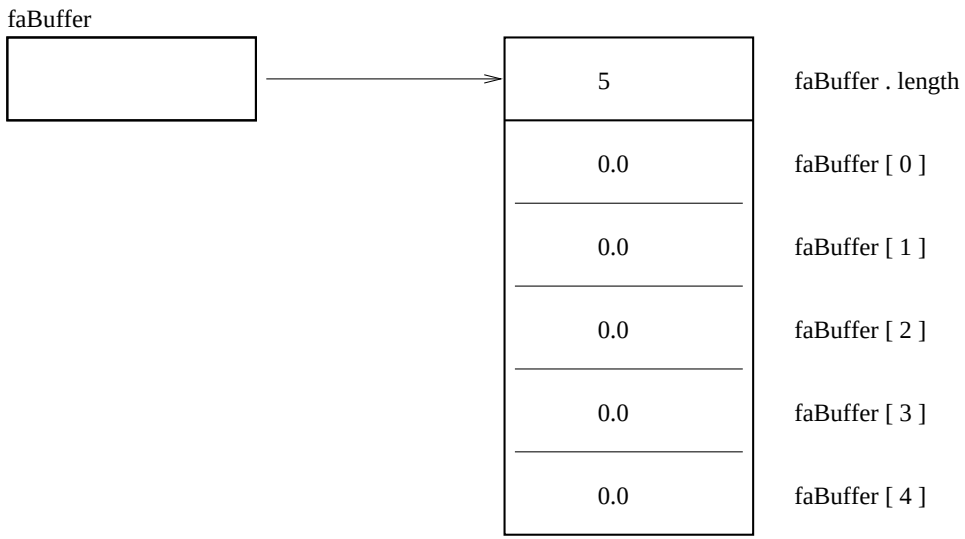
... individual items in the array are referenced by their position number in the array.

One-Dimensional Arrays

Example 1. Declaration for Array of Floating Point Numbers

```
float[] faBuffer = new float [5];
```

Layout of Memory



The first and last elements in the array are `faBuffer[0]` and `faBuffer[4]`.

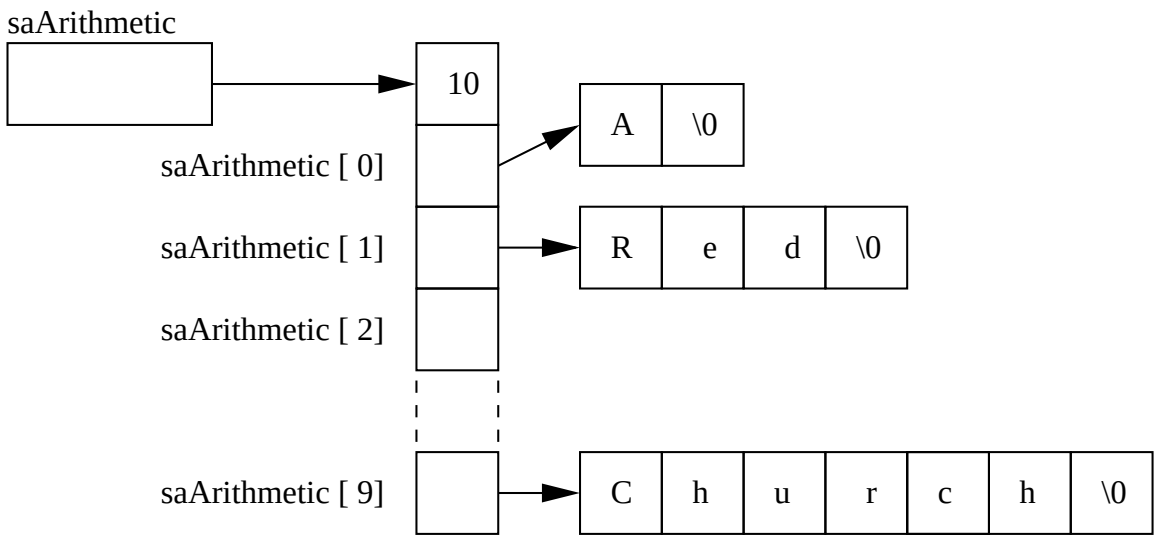
By default, all of the array elements will be initialized to zero!

One-Dimensional Arrays

Example 2. Declaration for Array of Character Strings

```
String [] saArithmetic = {   "A",  "Red", "Indian", "Thought",     "He",
                           "Might",  "Eat", "Toffee",      "In", "Church" };
```

Abbreviated Layout of Memory



Two-Dimensional Arrays

Multi-Dimensional Arrays

Multidimensional arrays are ...

... considered as arrays of arrays and are created by putting as many pairs of `[]` as of dimensions in your array.

Example 3. 4x4 matrix of doubles

```
double daaMat[][] = new double[4][4]; // This is a 4x4 matrix
```

Querying Dimensionality

You can query the different dimensions with the following syntax

```
array.length;          // Length of the first dimension.
array[0].length;       // Length of the second dimension.
array[0][0].length;    // Length of the third dimension.
.... etc ...
```

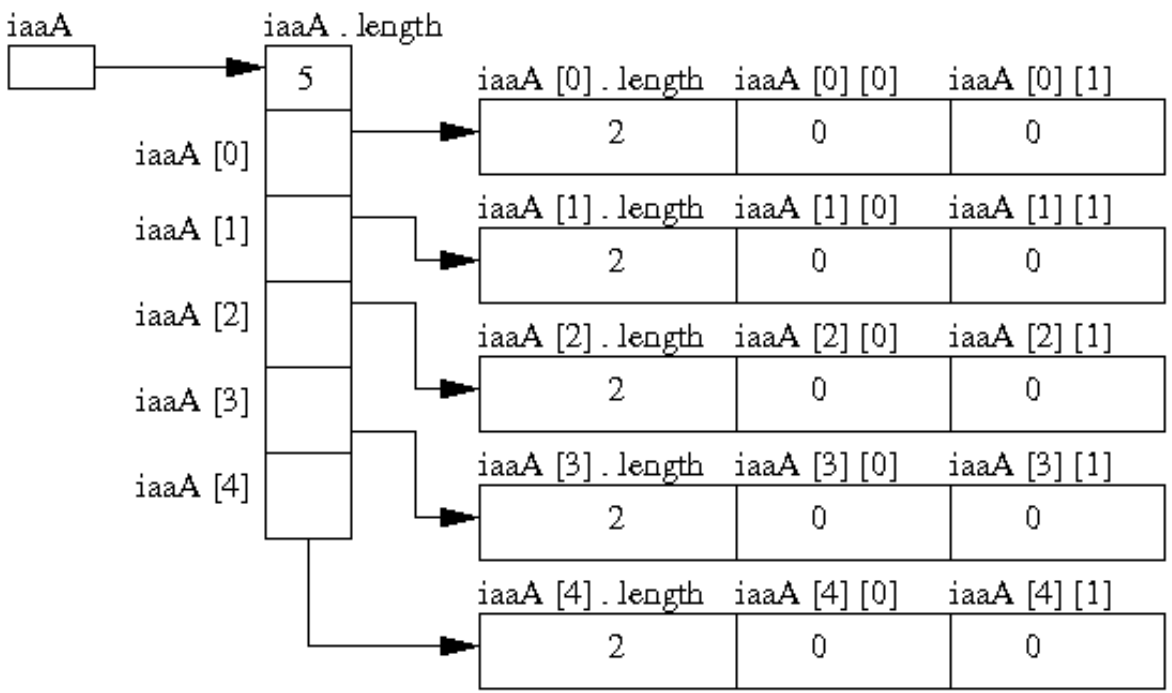
Two-Dimensional Arrays

Example 4. Two-Dimensional Array of Ints

Array Declaration

```
int [][] iaaA = new int [5][2];
```

Layout of Memory



Ragged Arrays

Skyline Matrix Storage – Savings in Required Memory

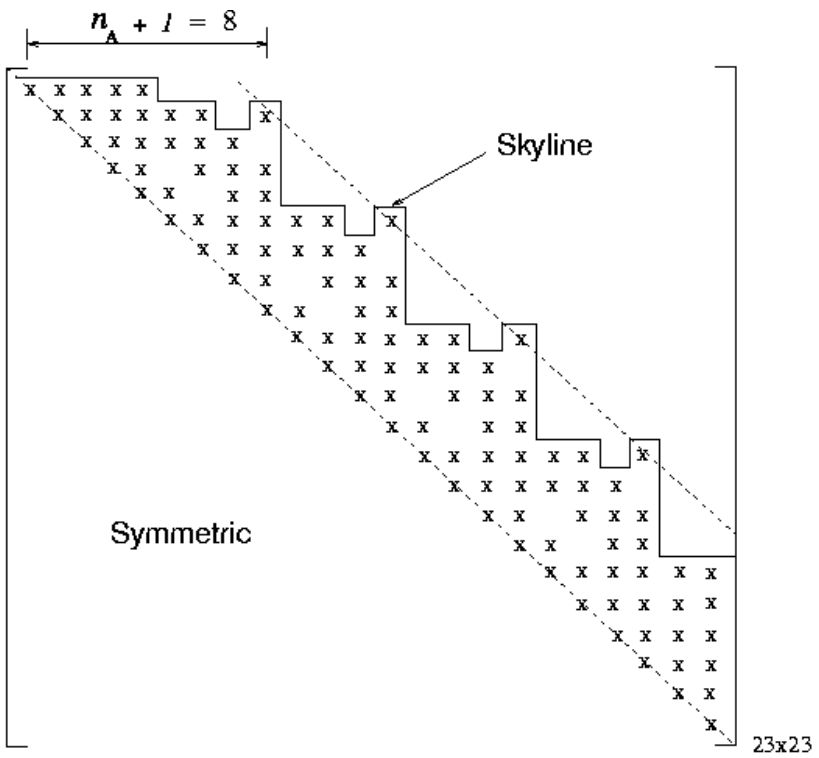


Figure 4.4: The structure of the system matrix $\mathbf{A}^t$ of our physics-based fish model, the skyline of $\mathbf{A}^t$ and its half-bandwidth $n_\mathbf{A}$. Each "x" represents a non-zero entries.

Ragged Arrays

Allocation Strategy 1 – Compiler Determines Layout of Memory

```
System.out.println("Test ragged arrays with variable row length");
System.out.println("Method 1: Compiler determines details");

int [][] iaaB = {{1,2},{3,4,5},{6,7,8,9},{10}};

System.out.println("");
System.out.println("No of rows      = " + iaaB.length );
System.out.println("Length of row 1 = " + iaaB[0].length );
System.out.println("Length of row 2 = " + iaaB[1].length );
System.out.println("Length of row 3 = " + iaaB[2].length );
System.out.println("Length of row 4 = " + iaaB[3].length );

System.out.println("Array: iaaB");
System.out.println("-----------");

for(int i = 0; i < iaaB.length; i=i+1) {
    for(int j = 0; j < iaaB[i].length; j=j+1)
        System.out.printf(" %3d ", iaaB[i][j] );
    System.out.printf("\n" );
}
```

Ragged Arrays

Allocation Strategy 1 – Output

```
Test ragged arrays with variable row length
-------------------------------------------
Method 1: Compiler determines details

No of rows      = 4
Length of row 1 = 2
Length of row 2 = 3
Length of row 3 = 4
Length of row 4 = 1
Array: iaaB
-----------
   1    2
   3    4    5
   6    7    8    9
  10
```

Ragged Arrays

Allocation Strategy 2 – Manual Assembly of Ragged Arrays

```
System.out.println("Method 2: Manual assembly of the array structure");

int[][] iaaC = new int[4][];    // Create number of rows...
iaaC[0] = new int[2];           // Create memory for row 1.
iaaC[1] = new int[3];           // Create memory for row 2.
iaaC[2] = new int[4];           // Create memory for row 3.
iaaC[3] = new int[1];           // Create memory for row 4.

iaaC[0][0] =  1; iaaC[0][1] = 2;
iaaC[1][0] =  3; iaaC[1][1] = 4; iaaC[1][2] = 5;
iaaC[2][0] =  6; iaaC[2][1] = 7; iaaC[2][2] = 8; iaaC[2][3] = 9;
iaaC[3][0] = 10;
```

Ragged Arrays

Allocation Strategy 2 – Print Output

```
System.out.println("Array: iaaC");
System.out.println("-----------");
for(int i = 0; i < iaaC.length; i=i+1) {
    for(int j = 0; j < iaaC[i].length; j=j+1)
        System.out.printf(" %3d ", iaaC[i][j] );
    System.out.printf("\n" );
}
```

Allocation Strategy 2 – Output

```
Method 2: Manual assembly of the array structure

Array: iaaC
-----------
   1    2
   3    4    5
   6    7    8    9
  10
```