



Arrays, vectors, and hashtables



Topics

- Object reference variables
- Arrays
- Unbalanced arrays
- Vectors
- Hashtable
- Collection classes
- Parameter passing

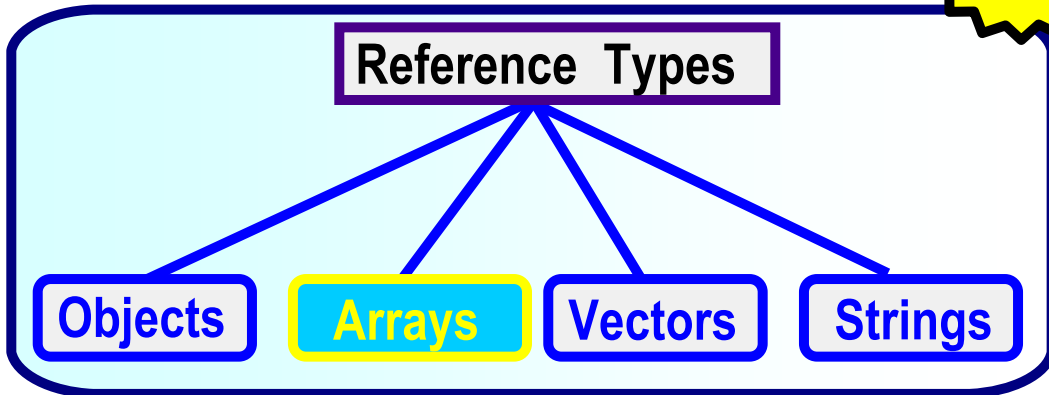
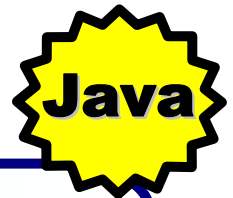
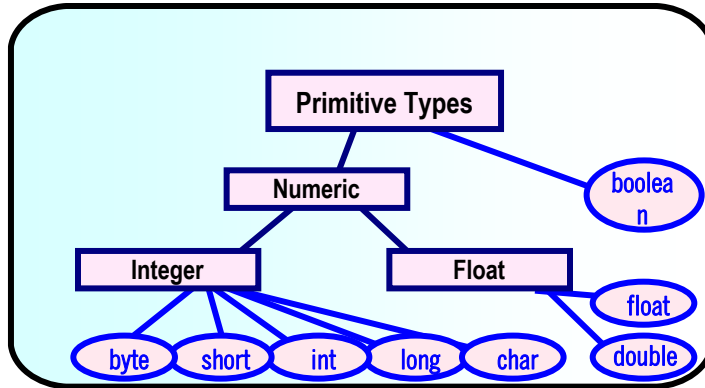


Unit objectives

After completing this unit, you should be able to:

- Declare, define, and access an array of objects or primitive types using Java syntax
- Define multidimensional and unbalanced, or non-rectangular arrays.
- Use the vector and hashtable JDK collection classes
- Describe how the Java Virtual Machine passes parameters between methods

Reference data types



Reference types

- Reference types can be declared as one of the following:
 - **Class type**: This can be any of the classes Java supplies in its `java.xxxx` packages, or one of your own classes.
 - **Interface type**: Interfaces are special kinds of classes that define only method names and parameters (signatures) but not code. They are covered in Chapter 9.
 - **Array type**: Arrays are declared by using [and] brackets, one pair per dimension, along with the primitive or reference type each element of the array is to be of.
- As you recall, objects are "instances" of classes (or arrays).
 - Instantiated with `new` operator

Object references

- Operations allowed on object ref variables:
 - Access instance variables inside the object, using dot notation.
Example:
 - `myObject.limit = 100;`
 - Invoke methods on the object, using dot notation:
 - `myObject.setLimit(100);`
 - Cast between object types (Chapter 9).
 - Use instanceof operator to verify class type (Chapter 9).
 - Compare or set to *null*, a keyword in Java.
 - Compare memory addresses, using `==` , `!=` or the conditional operator `?:`, as in:
 - `myNewObject =`
 - `(myObject != null) ? myObject : myOtherObject;`

Arrays



► Array types:

✓ One-dimensional

✓ Tables

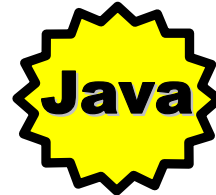
✓ Dynamic APIs

► Initializing:

✓ Compile time

✓ Pre-Runtime

✓ Runtime



► Array types:

✓ One-dimensional

✓ Multidimensional

✓ Hashtable class

✓ Vector class

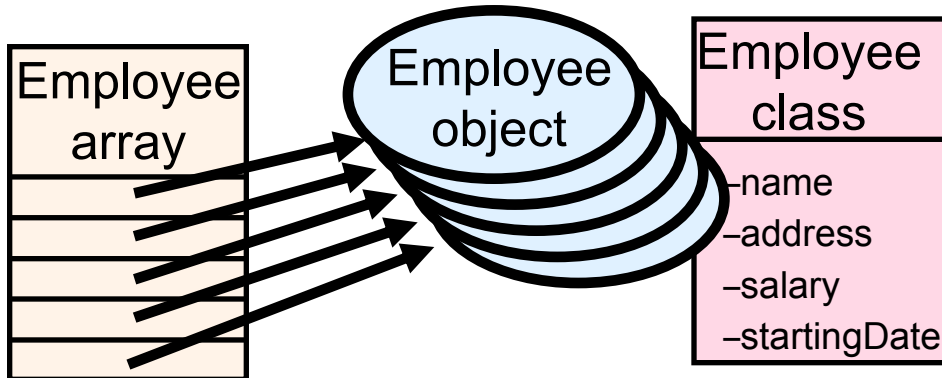
► Initializing:

✓ Compile time

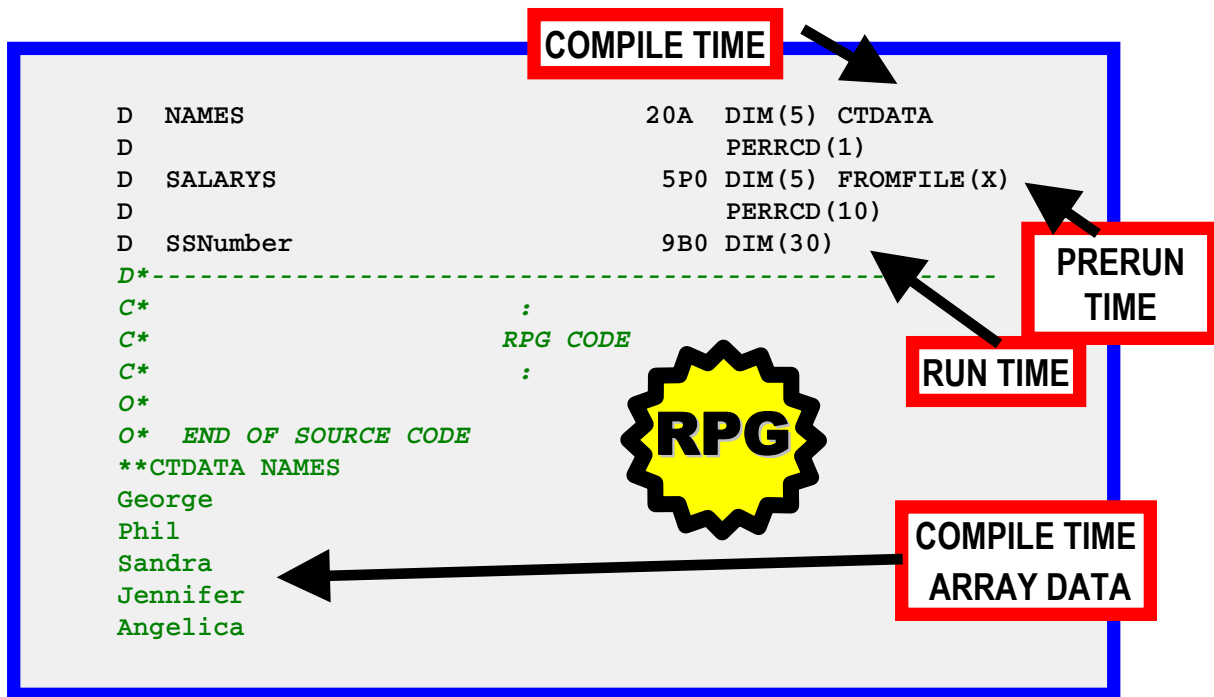
✓ Runtime

What about MODS?

- What about Multiple Occurring Data Structures (MODS)?
 - In RPG, these are arrays of structures.
 - In Java, these are arrays of objects.
 - The object's class = the DS in RPG.



Arrays in RPG

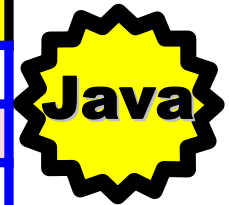


Introduction to arrays

- Surprise! In Java, arrays are objects!
 - Array variables are object references to an array object.
 - Array objects need to be instantiated like all objects.



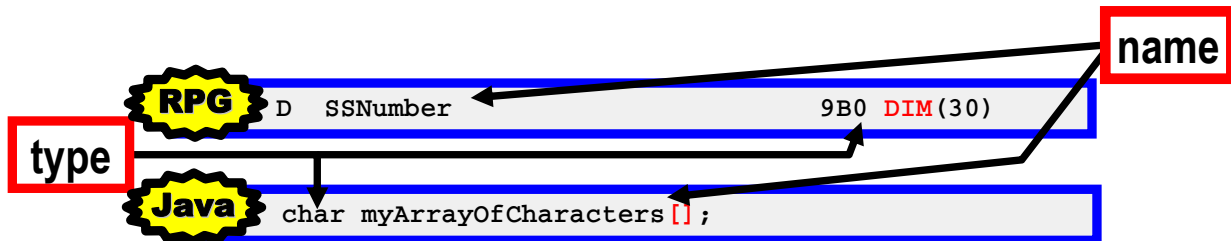
RPG	Java
Declare the array	Declare the array
	Instantiate the array
Initialize the array	Initialize the array
Use the array	Use the array



- Arrays are "special" objects in Java.
 - No class type
 - Special operator [] for accessing elements
 - Can optionally instantiate and initialize in one step without using **new** operator

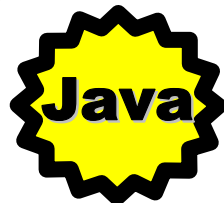
Declaring arrays

- RPG supports single dimension arrays.
 - They are defined using the DIM keyword.
 - The size of the array (max elements) is the DIM parm.
- Java supports multiple dimension arrays.
 - They are defined using bracket pairs: []
 - One pair per dimension!
 - The size of the array is specified later, at creation time.
- In both RPG and Java, you define the data type for the elements, and a name for the array variable.



Declaring arrays: More

- Note that the brackets can be either after the type, or after the name:
 - `char[] myArrayOfChars;`
 - `char myArrayOfChars[];`
- For multiple dimensions use multiple pairs:
 - `char arrayOfBuffers[][];` // two dimension
 - `int[] rowColumn[];` // two dimension
 - `int rowColumnDepth[][][];` // three dimension
- In Java and RPG all array elements must be of the same type
 - For example, all characters or all integers or all...
- In Java array elements can be objects:
 - `MyClass arrayOfObjects[];`

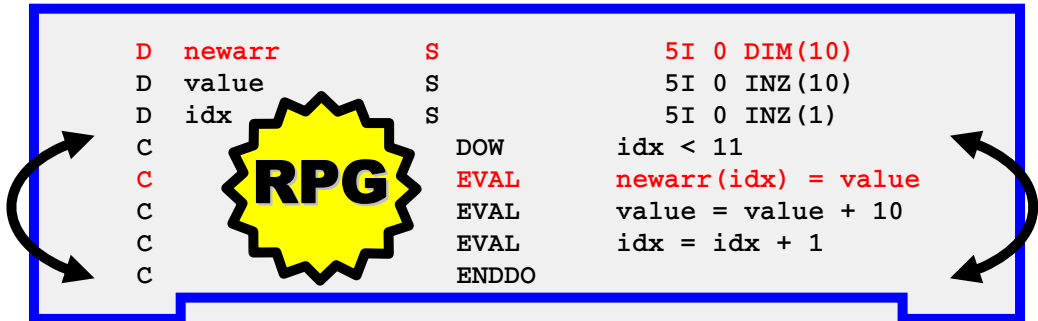


Instantiating arrays

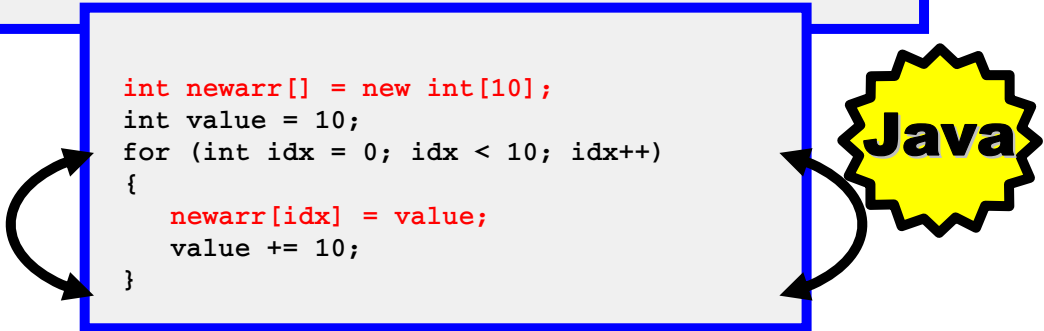
- After declaring, you create the array.
 - Use **new** operator, followed by element type.
 - Use brackets [] with **new**, not parens ()
 - Specify the array size here, as with **DIM** in RPG:
 - `char[] myArrayOfChars;`
 - `myArrayOfChars = new char[100];`
- For multiple dimensions use multiple pairs:
 - `char arrayOfBuffers[][]; // two dimension`
 - `arrayOfBuffers = new char[10][1000];`
- Can be done in two steps, or just one:
 - `char myArrayOfChars[] = new char[100];`
- Before instantiating, array vars are "null":
 - `if (myArrayOfChars == null)`
 - `myArrayOfChars = new char[100];`

Run-time initialization

- In RPG and Java, you can initialize arrays dynamically (at *runtime*):
 - Typically in a *loop*:



```
D newarr      S      5I 0 DIM(10)
D value      S      5I 0 INZ(10)
D idx        S      5I 0 INZ(1)
C
C      DOW      idx < 11
C      EVAL    newarr(idx) = value
C      EVAL    value = value + 10
C      EVAL    idx = idx + 1
C      ENDDO
```



```
int newarr[] = new int[10];
int value = 10;
for (int idx = 0; idx < 10; idx++)
{
    newarr[idx] = value;
    value += 10;
}
```

Run-time initialization notes

- Important notes about array element indexing in Java versus RPG:
 - 1. Use [idx] in Java versus (idx) in RPG:

C

RPG

EVAL

newarr(**idx**) = value

Java

newarr[**idx**] = value;

- 2. Indexing starts at zero ("zero-based") in Java and one ("one-based") in RPG:

D idx

S

5I 0 INZ(**1**)

C

DOW

idx < **11**

Java

for (int idx = **0**; idx < **10**; idx++)

- 3. "Length" is a built-in variable that all arrays have:

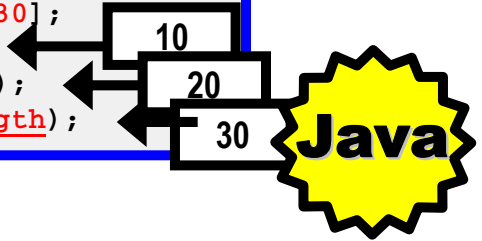
Java

for (int idx = 0; idx < newarr.length; idx++)

Initializing multi-dimensional arrays

- “Length” for dimension “n” is used with “n-1” brackets

```
int array[][][] = new int[10][20][30];  
System.out.println(array.length);  
System.out.println(array[0].length);  
System.out.println(array[0][0].length);
```



- Nest **for** loops, one per dimension
 - Outer-most loop is first dimension, inner is last

```
int rtArray[][] = new int[3][3]; // Two dim array  
  
for (int xIdx=0; xIdx < rtArray.length; xIdx++)  
    for (int yIdx=0; yIdx < rtArray[xIdx].length; yIdx++)  
        ...
```


Arrays of objects

- Array elements can be objects.
 - Create each element as part of initializing.



```
MyClass myArray[] = new MyClass[10]; // array of objects

for (int xIdx=0; xIdx < myArray.length; xIdx++)
    myArray[xIdx] = new MyClass(); // populate with new obj
```

Versus

```
int myArray[] = new int[10]; // array of ints

for (int xIdx=0; xIdx < myArray.length; xIdx++)
    myArray[xIdx] = 10 * xIdx; // populate with new number
```

Multidimensional initialization example

```
class TestMultiArrayRT
{
    public static void main(String args[])
    {
        int rtArray[][] = new int[3][3]; // Two dim array
        int value = 1;
        // Loop through all rows...
        for (int xIdx=0; xIdx < rtArray.length; xIdx++)
        {
            // Loop through all columns...
            for (int yIdx=0; yIdx < rtArray[xIdx].length; yIdx++)
            {
                rtArray[xIdx][yIdx] = value++; // assign and incr't
                System.out.print(rtArray[xIdx][yIdx] + " ");
            } // end inner for loop
            System.out.println();
        } // end outer for loop
    } // end main method
} // end TestMultiArrayRT class
```



1 2 3

4 5 6

7 8 9

Setting array size

- In both RPG and Java, when an array is created, its size is fixed.
 - It cannot be resized.
 - Java allows deferring creation (using `new`) until after size has been determined.
 - Call **new** after determining how big array needs to be.
 - You can use any expression for the array size.
 - It is evaluated at runtime, not compile time as it is in RPG.
 - The power of objects!

Compile-time initialization (1 of 2)

- RPG allows initializing at declaration time (*compile time*):



```
D  employee
D
**CTDATA employee
ABCDEF
GHIJKL
```

```
3A  DIM (4)  PERRCD (2)
      CTDATA
```

```
employee(1) = 'ABC'
employee(2) = 'DEF'
employee(3) = 'GHI'
employee(4) = 'JKL'
```

D-Spec Keyword

Explanation

DIM

All RPG arrays must specify this

PERRCD

Number of elements per record in the file

CTDATA

Indicates initialization data is in CTDATA records at bottom of file. Same name as array field name

Compile-time initialization (2 of 2)

- Java allows initializing at declaration time (*compile time*):

```
String employee[] = {"ABC", "DEF", "GHI", "JKL"};
```

Note: *String* objects are covered in Chapter 7.

```
employee[0] = "ABC"  
employee[1] = "DEF"  
employee[2] = "GHI"  
employee[3] = "JKL"
```



Java

- Special Java syntax:
 - Values specified between braces
 - Semi-colon needed after last brace
 - Values for each element separated by commas
 - No need to use new operator (implied)

Compile time initializing of multidimensional arrays

- Complete set of {v,v,v} values per dimension.
- The "v" values for each dimension except leftmost is complete set for another array.
 - Multidimensional arrays are arrays of arrays!

```
int ctArray[] = { 0, 1, 2 };
```

one dimension:
nesting == 1

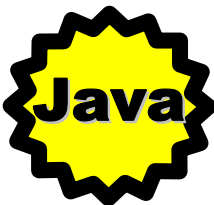
```
int ctArray2[] [] =  
{  
    {10,11,12},  
    {30,31,32}  
};
```

$2 * 3$

two dimensions:
nesting == 2

```
int ctArray3[] [] [] =  
{ { { 0, 1, 2}, { 3, 4, 5} },  
  { {10,11,12}, {13,14,15} },  
  { {20,21,22}, {23,24,25} }  
};
```

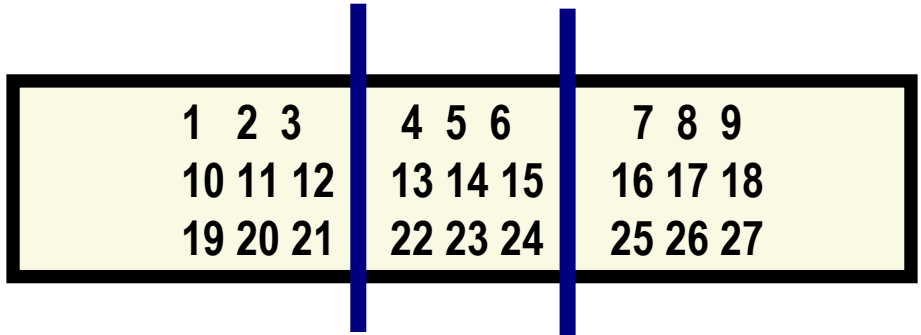
three dimensions:
nesting == 3



$3 * 2 * 3$

Exercise

- Declare a 3-dimensional array of integers and initialize it to:



1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27

- Then print it.

Solution



```
public static void main(String args[])
{
    // declare and initialize
    int ctArray[][][]
        = { { { 1 , 2 , 3 }, { 4 , 5 , 6 }, { 7 , 8 , 9 } },
            { { 10, 11, 12}, { 13, 14, 15}, { 16, 17, 18} },
            { { 19, 20, 21}, { 22, 23, 24}, { 25, 26, 27} } };
    // print each value...
    for (int xIdx=0; xIdx < ctArray.length; xIdx++)
    {
        for (int yIdx=0; yIdx < ctArray[xIdx].length; yIdx++)
        {
            for (int zIdx=0; zIdx < ctArray[xIdx][yIdx].length;
                zIdx++)
            {
                System.out.print(ctArray[xIdx][yIdx][zIdx] + " ");
            } // end inner for loop
            System.out.print("\n");
        } // end middle for loop
        System.out.println();
    } // end outer for loop
} // end main method
```

1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18
19	20	21	22	23	24	25	26	27

Arrays are references

- What does it mean to assign one array to another?
 - `array1 = array2;`
- Assignments are not array copies.
 - Remember, arrays are objects.
 - Assigning only copies the array (object) memory address.
- After assignment, all changes and references to first array affect and reflect the second array and vice versa.
 - Two array references point to same array in memory.

Array reference example

Java

```
int arrayOne[][] = { { 1 , 2 }, { 3 , 4 } };
int arrayTwo[][] = { { 5 , 6 }, { 7 , 8 } };
int tempArray[][];

tempArray = arrayOne;
for (int xIdx = 0; xIdx < tempArray.length; xIdx++)
{
    for (int yIdx = 0; yIdx < tempArray[xIdx].length; yIdx++)
        System.out.print(tempArray[xIdx][yIdx] + " ");
    System.out.println();
}

tempArray = arrayTwo;
for (int xIdx = 0; xIdx < tempArray.length; xIdx++)
{
    for (int yIdx = 0; yIdx < tempArray[xIdx].length; yIdx++)
        System.out.print(tempArray[xIdx][yIdx] + " ");
    System.out.println();
}
```

tempArray is not created. It will point to other arrays

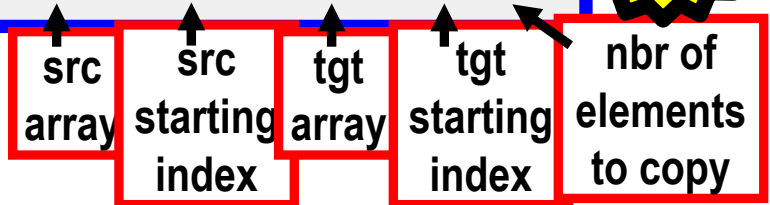
1 2
3 4

5 6
7 8

Copying arrays

- What if you do want to copy contents of one array to another?
Three choices:
 - Loop through both arrays, copying elements of one array to another.
 - Use clone() method that all arrays have.
 - Use method arraycopy in System class (for single dimension arrays only).

```
int arraySrc[] = {1,2,3,4};  
int arrayTgt[] = new int[4];  
System.arraycopy(arraySrc, 0, arrayTgt, 0, 4);
```



Pre-runtime arrays

- RPG has pre-runtime arrays.
 - Load values from a file, save values to a file
- Java does *not* have this support built-in.
 - But you could write your own.

```
FSTATE      UF      F      70      DISK
D STATEARR          S          2A      DIM(20)
D                                     PERRCD(1)
D                                     FROMFILE (STATE)
D                                     TOFILE (STATE)
```

A yellow starburst logo with the letters "RPG" in black, positioned in the center of the code block.

D-Spec Keyword

Explanation

DIM

Specify the number of dimensions.

FROMFILE(filename)

Read from this file at program load time.

PERRCD

Number of elements per record in the file.

TOFILE(filename)

Write to this file at termination of the program.

Unbalanced arrays

- Usually multidimensional arrays are rectangular.
 - For example, three rows, each with four columns
- But they do not have to be.
 - Multidimensional arrays in Java are arrays of arrays.
 - And arrays are objects!
 - So each dimension's elements can be different lengths
- This is called unbalanced arrays.
 - Rule is to leave leftmost length unspecified.

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



- The following is invalid balanced array:

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

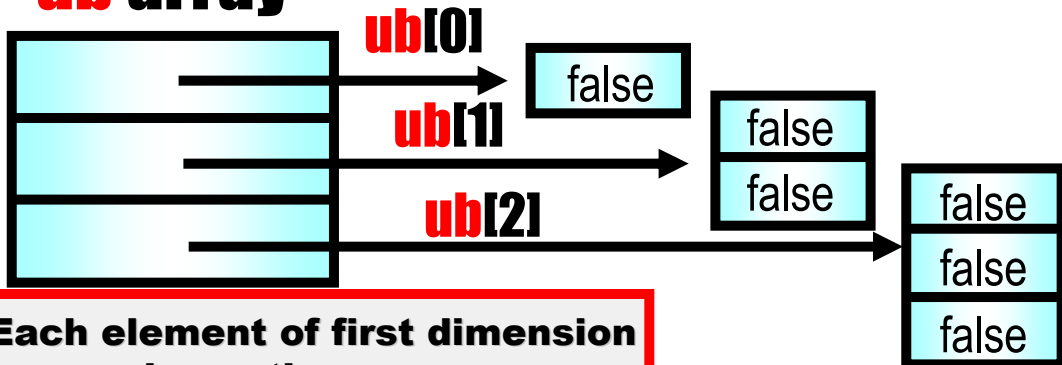


Unbalanced array example

```
boolean ub[] [] = new boolean[3] [];  
for (int idx = 0; idx < ub.length; idx++)  
{  
    ub[idx] = new boolean[idx+1];  
}
```

**Indicates
unbalanced
array**

ub array



**Each element of first dimension
is another array.**

Java.util.Arrays

- New as of JDK 1.2.0 is a helper class named Arrays in package java.util:
 - Numerous static methods for common array operations
- Methods include:
 - `binarySearch`
 - Efficiently search a sorted array for a given element.
 - `Equals`
 - Compare two arrays for equality.
 - `Sort`
 - Sort all the elements in an array.
 - `Fill`
 - Initialize all element in an array to same value.

Introduction to vectors

- In both RPG and Java, when an array is created, its size is fixed.
 - It cannot be resized.
 - This leads to creating arrays large enough to hold the maximum number of elements.
 - Can be a big waste of memory.
- Java supplies a solution: *vectors*.
 - Vector is a class in java.util package.
 - To use, need `"import java.util.*;"` at top of file.
- Vectors are like dynamically sizable arrays.
 - Do not need to specify initial size.
 - Size grows as needed when items are added.

Using vectors

- Using vectors:
 - Create empty vector by instantiation.
 - Add items using addElement method.

```
import    java.util.*;

Vector myVector = new Vector(); // create vector
String inputString = getNextInput(); // some method
while (inputString != null)
{
    myVector.addElement(inputString); // populate
    inputString = getNextInput();
}
```

- Query number of elements using size method.
- Query specific element using elementAt method.

```
for (int idx = 0 ; idx < myVector.size() ; idx++)
    System.out.println(myVector.elementAt(idx));
```

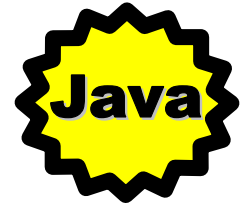
...more

Vector methods

Method	Description
addElement(Object)	Adds specified object to the end of this vector, increasing its size by one.
capacity()	Returns the current capacity of this vector. Capacity is the potential size of a vector, versus its actual size or element
clone()	Returns a copy of this vector. All elements are copied.
contains(Object)	Tests if the specified object is an element in this vector.
copyInto(Object[])	Copies the elements of this vector into the specified array.
elementAt(int)	Returns the element at the specified index (zero based).
setElementAt(Object, int)	Sets the element at given integer index (zero based) to given object.
elements()	Returns an enumeration of the elements of this vector. See chapter nine for information on enumerators.
ensureCapacity(int)	Increases the capacity of this vector to the given number.
firstElement()	Returns reference to the first element of this vector.
indexOf(Object)	Searches for the first occurrence of the given parameter, testing for equality using the equals method on each
indexOf(Object, int)	Searches for the first occurrence of the first parameter, beginning the search at the zero-based index specified in the second parameter. Again, tests for equality using equals. Returns -1 if not found.

Hashtables (1 of 2)

- Java supplies a class for simple lookup tables
 - Hashtable in package `java.util`
 - Contains pairs of objects
 - A key object and a value object
 - Objects can be of any class
 - Use `put` to insert, `get` to retrieve



insert by key,
value pair

```
Hashtable customers = new Hashtable();  
customers.put("011002", "Phil Company");  
customers.put("110034", "George Limited");  
. . .
```

```
String georgeEntry = customers.get("110034");
```

search by
key, get
value

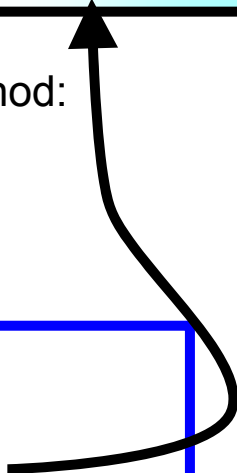
Hashtables (2 of 2)

- Hashtable stores items in name/value pairs.
 - The name is the key.

011002	Phil Company
110034	George Limited

- To get a list of the values, use the elements method:
 - Returns an object of type Enumeration
 - Use `hasMoreElements` and `next` to walk it

```
Enumeration names = customers.elements();  
while (names.hasMoreElements())  
    System.out.println(names.next());
```



Hashtable methods

Method	Description
<code>clear()</code>	<i>Clears all key/value pairs from the table</i>
<code>clone()</code>	<i>Returns a copy of this table. All elements are copied.</i>
<code>containsKey(Object)</code>	<i>Tests if the specified key is an element in this table.</i>
<code>containsValue(Object)</code>	<i>Tests if the specified value is an element in this table</i>
<code>elements()</code>	<i>Returns an enumeration of the values of this table.</i>
<code>get(Object)</code>	<i>Returns the value of the element with the given key (or null).</i>
<code>keys()</code>	<i>Returns an enumeration of the keys of this table.</i>
<code>isEmpty()</code>	<i>Tests if this table is empty or not.</i>
<code>put(Object, Object)</code>	<i>Inserts a key and value pair. If key already exists, value is replaced.</i>
<code>remove(Object)</code>	<i>Removes element with the specified key.</i>
<code>size()</code>	<i>Returns number of elements in this table.</i>

Casting is required

- Vector and hashtable are defined by Java to hold objects of class type Object.
 - Special class that all classes "inherit" from
 - We cover inheritance in Chapter 9.
- This means to get a value out of a vector or hashtable you must cast:
 - You, objects can be cast too (Chapter 9)
 - For example:
 - Customer cust1 = (Customer)custtable.get("011012");
 - Else, it will not compile.

Important methods

- When putting your own classes inside arrays, vectors, and hashtables, there are some important methods that your class needs to supply:
 - `boolean equals(Object compareToObject)`
 - Java calls this when searching for given element.
 - Will pass as a parameter the object to compare to.
 - You decide if they are equal (for example, employee IDs match).
 - `int hashCode()`
 - Java calls this in hashtables when searching for key.
 - It should return `hashCode()` of "key" instance variable, such as employee ID for an Employee class.

New collection classes

- 1.2 also added support for "collections"
 - **Classes for storing lists of items**
 - You can decide on most efficient choice depending on your list's attributes
 - **For example:**
 - Key-value pairing requiring lookup? "Map"
 - Ordered collection with duplicates? "List"
 - Ordered collection without duplicates? "Set"
 - Sorted ordered coll'n without dupes? "SortedSet"
 - **Interfaces and classes added to java.util package:**
 - **Maps:**
 - Map interface, HashMap, Hashtable, WeakHashMap, TreeMap classes
 - **Lists:**
 - List interface, LinkedList, Vector, ArrayList classes
 - **Sets:**
 - Set interface, HashSet class
 - **SortedSets:**
 - SortedSet interface, TreeSet class

New 1.2.x collections

Package	Description
<code>java.util.ArrayList</code>	Resizable array, similar to vector
<code>java.util.Arrays</code>	Static helper methods for arrays, such as sorting
<code>java.util.Collections</code>	Static helper methods for collections, such as sorting
<code>java.util.HashMap</code>	For key-value mapping tables that contains duplicates
<code>java.util.HashSet</code>	For key-value mapping tables that do not contain duplicates
<code>java.util.LinkedList</code>	For doubly-linked lists
<code>java.util.Stack</code>	A LIFO stack
<code>java.util.TreeMap</code>	For "red-black" trees holding for keyed ascending key-value mappings. Log(n) time for search, get, put and remove operations
<code>java.util.TreeSet</code>	For sorted key-value mapping tables that do not contain duplicates. Log(n) time for search, get and put operations
<code>java.util.WeakHashMap</code>	Keys may be removed by garbage collector

Parameter passing (1 of 2)

- RPG procedure parameters are passed by reference by default.
 - Address of parameter passed versus copy of value.
 - Changes made to parameter in procedure are reflected in calling code.
 - For read-only parameters, specify keyword VALUE on procedure parameter prototype.
- Java method parameters are passed by value only.
 - For primitive parameters, copy of value is passed.
 - For object parameters, copy of object memory address is passed.

Parameter passing (2 of 2)

- Pass by value means
 - Changes made to parameter are not reflected in caller code:

```
public static void main(String args[])
{
    int value1 = 10;
    int value2 = 20;
    System.out.println(value1 + ", " + value2);
    swap(value1, value2);
    System.out.println(value1 + ", " + value2);
} // end main

public static void swap(int parm1, int parm2)
{
    int temp = parm1;
    parm1 = parm2;
    parm2 = temp;
} // end of method swap
```

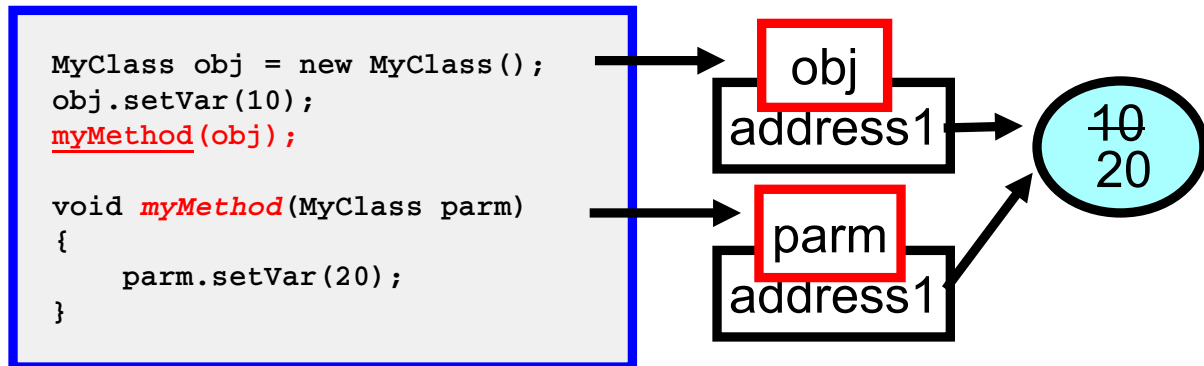


10, 20

10, 20

Passing objects

- When passing an object as a parameter:
 - Memory address of object is passed by value so that you cannot change which object the variable points to.
 - However, changes made to the object via method calls or direct access are reflected in caller's code.



Parameter passing

- What if you want to change a parameter?
 - Pass array elements instead. Arrays are objects.
 - Changes made to array elements are effective.

```
public static void main(String args[])
{
    int value1[] = {10};
    int value2[] = {20};
    System.out.println(value1[0] + ", " + value2[0]);
    swap(value1, value2);
    System.out.println(value1[0] + ", " + value2[0]);
} // end main

public static void swap(int parm1[], int parm2[])
{
    int temp = parm1[0];
    parm1[0] = parm2[0];
    parm2[0] = temp;
} // end of method swap
```

10, 20

20, 10

Topics covered

- Object reference variables
- Arrays
- Unbalanced arrays
- Vectors
- Hashtable
- Collection classes
- Parameter passing



Unit summary

Having completed this unit, you should be able to:

- Declare, define, and access an array of objects or primitive types using Java syntax
- Define multidimensional and unbalanced, or non-rectangular arrays
- Use the vector and Hashtable JDK collection classes
- Describe how the Java Virtual Machine passes parameters between methods